Aalto University
School of Engineering

Animesh Kumar

Lane Following using Behavioural Cloning

Master's Thesis
Aalto University
School of Engineering
Department of Mechanical Engineering

Thesis submitted as partial fulfilment of the requirements for the degree of Master of Science in Technology

Espoo, 24 July 2019
Supervisor: Professor Kari Tammi
Advisor: Professor Kari Tammi

Aalto University
School of Engineering

Aalto University, P.O. BOX 11000, 00076
AALTO
www.aalto.fi

Abstract of master's thesis

Author Animesh Kumar		
Title of thesis Lane Following using Behavioural Cloning		
Master programme Master's Programme in Mechanical Engineering		**Code** ENG25
Thesis supervisor Professor Kari Tammi		
Thesis advisor(s) Professor Kari Tammi		
Date 24.07.2019	**Number of pages** 59+11	**Language** English

Abstract

With the rise in the research relating to Artificial Intelligence along with the growing concern of everyday road accidents due to human error, the research pertaining to Autonomous Vehicles has been soaring to new highs. However, this technology in its current form has serious limitations such as restricted use during adverse conditions (such as snow), inability to identify manual traffic instructions, abnormal traffic behaviours etc. This is one of the reasons that even the vehicles with most autonomous features, exhibit only a Level 2 or Level 3 of driving automation. Hence, in order to reach further levels of automation, it may be useful to create a symbiotic technology between autonomous vehicles and traffic control models. This thesis work will work as an elementary stepping stone to create such a symbiosis by identifying a Lane Following Model using Convolutional Neural Networks. Specifically, a Behavioural Cloning Model along with a Road Classification Model is developed in order to mimic human driving characteristics which ideally works independent of lane markings and to regulate this driving characteristics by reading road signs with satisfactory levels of accuracy.

Keywords Autonomous Vehicles, Lane Following, Convolutional Neural Network, Deep Learning, End to End Learning, Behavioural Cloning, NVIDIA architecture, Road Symbol Classification

Foreword

I would like to take this opportunity to thank my supervisor, Professor Kari Tammi, for his constant support and guidance. He has been pivotal in ensuring that the work done in the thesis is upto acceptable standards for a master's degree. I would also like to extend my thanks to Jari Vepsäläinen for his encouraging words and also his guidance regarding the scope of the thesis. Risto Ojala also played a crucial role in clearing doubts pertaining to the topic, especially during the initial stages of study, when the research topic was relatively alien to me.

I would also like to acknowledge Henry Ford Foundation, for supporting the thesis with their valuable funding, thereby enabling a focused research on the topic.

Finally, I would like to thank my parents as well as my brother for their constant support.

Espoo 24.07.2019

Animesh Kumar

Table of Contents

Chapter 1: Introduction

The world, in general, is moving towards a more autonomous future, and autonomous vehicles are an integral part in making the step towards it. With the rise in the research relating to Artificial Intelligence along with the growing concern of everyday road accidents due to human error, the research pertaining to Autonomous Vehicles has been soaring to new highs. According to Dixit et al., authors of the article 'Autonomous Vehicles: Disengagements, Accidents and Reaction Times' [1], with the advancements in technology and with the increase in the number of miles driven by the autonomous vehicle the reaction time for the vehicles to aptly respond to an accident prone situation has been on the decline.

1.1 Problem Overview

In its current form, the self-driving technology has serious limitations such as restricted use during adverse conditions (such as snow), inability to identify manual traffic instructions, abnormal traffic behaviours etc. This is one of the reasons that even the vehicles with most autonomous features, exhibit only a Level 2 or Level 3 of driving automation. For the time being, even the most autonomous vehicles running on roads can only be categorized to somewhere between a Level 2 or Level 3 [2]. In fact, most vehicles with autonomous features have only recently started obliging to the need to create a highly automated vehicle given the growing concerns regarding road accidents and environmental repercussions of a human-driven vehicle. Several automotive companies have even predicted that to obtain a Level 5 automation, one might have to wait for up to 20 more years, despite the continuous research and effort being into this field. The following is a brief description of the various levels of driving automation as described by SAE International [2]:

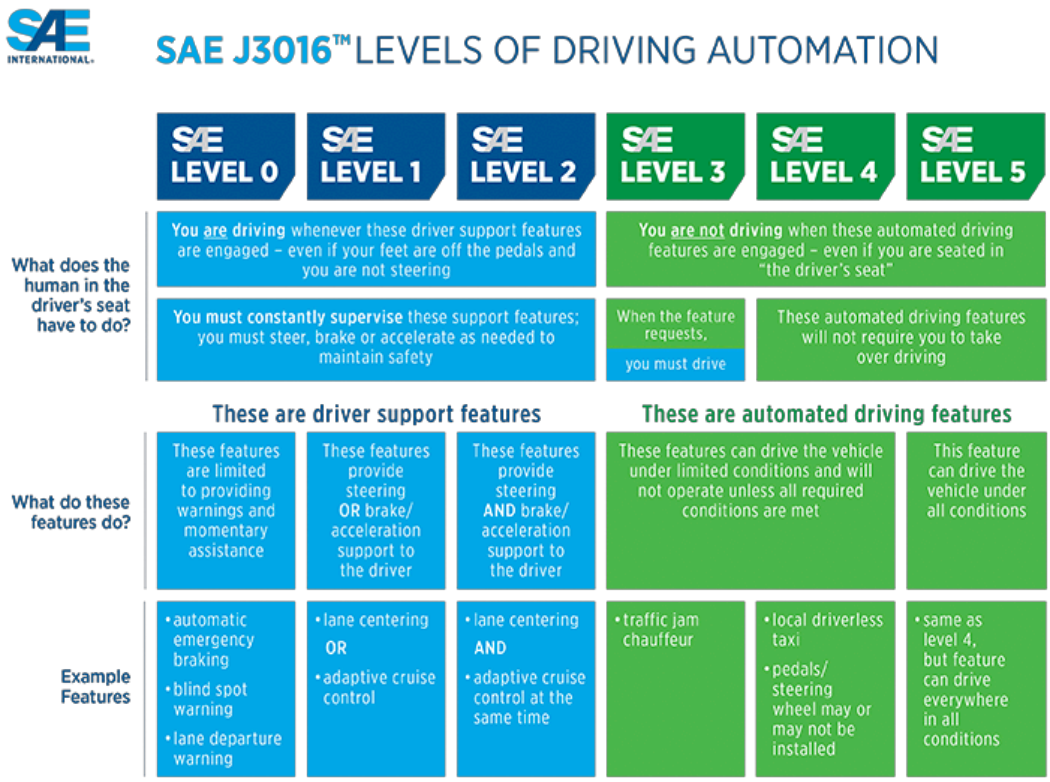


Figure 1: Levels of driving automation by SAE International [2]

Apart from the technological difficulties, the fact that in most countries the legislation is lagging to inhibit operations of existing autonomous vehicles is also a limiting factor. For example, in India, the Motor Vehicles Act of 1988 [3] accounts for a liable person responsible for the vehicle. However, in case of an autonomous vehicle, it would be a complex task to implement this law. The legislation hence can be benefitted by creating special regulations and amendments for the autonomous vehicles in traffic conditions. Hence, it might be a suitable prospect if the existing traffic flow control models are linked with autonomous vehicles and its advancements. Potentially, this would not only facilitate the existing technologies of the autonomous vehicles to obtain data for current traffic en-route but also create a suitable action plan by analysing the traffic behaviour and pattern, hence preventing probable abnormal driving behaviours, adverse conditions etc. This hence would also help in establishing regulations regarding autonomous vehicles such as separate lanes or even make temporary regulations such as autonomous driving only during certain hours. This may be the push that the industry requires at the moment to accelerate the advancements. Also, in an advanced form, this can be used to implement autonomous vehicles in highly populated markets with arbitrary driving style (non-lane following models) such as India.

1.2 Objective

As a preliminary step towards building this symbiotic technology, a dynamic vehicular model can be built that can be utilised as a link between the traffic flow and autonomous vehicle technologies. However, the biggest challenge in developing such a model would be deciding the methodology as the model would need to be futureproofed in order to encapsulate the developing technologies on both fronts. Also, to act as a link, the model needs to share a common feature with both the technologies. To tackle this, a deeper look into what the existing driving automation model and traffic models are based on can be taken into consideration.

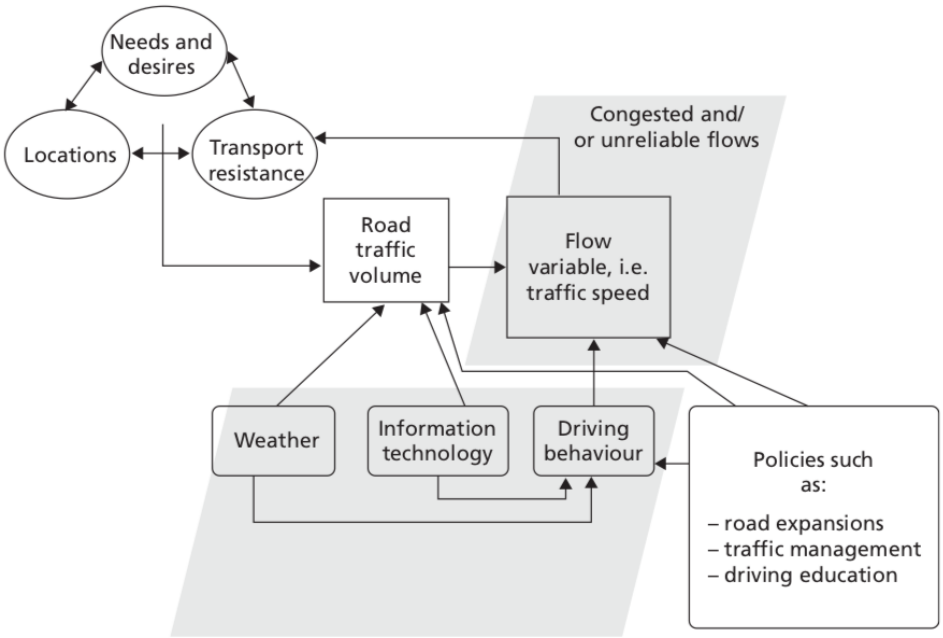


Figure 2: Sample Traffic Flow Model [4]

As can be observed from the block diagram above by, the driving behaviour shows the potential to be utilised as the desired link. The driving behaviour of an autonomous car is determined by the algorithm used in the controller that interpret the signals from the sensors attached in the car and correspondingly drives the throttle, steering, brakes amongst various actuators to attain simple objectives such as following a lane.

Hence, the objective of the thesis is to develop a lane following algorithm using a Convolutional Neural Network (CNN) to enable behaviour cloning. With the help of behaviour cloning, one can enable mimicking the behaviour of a human driver through CNN's. As human drivers are equipped to handle the vehicle in any traffic, weather or light conditions, hence training an autonomous vehicle model to mimic these abilities would suffice the driving behaviour characteristics required.

1.3 Scope

The scope of the thesis will be limited to developing a system that shows a substantial resemblance to the driving behaviour of a human driver in both lane marked roads as well as roads with no lane markings. In order to achieve this, simulation-based testing was performed in an environment where no lanes were marked around a circuit. The model was expected to extract visual features and steer the simulated vehicle around the circuit, after being trained by a human driver. The techniques to be investigated were limited to the domain of Traditional Computer Vision and Deep Learning. For the purposes of data collection, prebuilt simulation tracks were utilised, instead of customised tracks.

The thesis will be divided into six chapters, including the introduction. In the next chapter, the various possible methods for achieving a lane following model that fits the description are discussed and the choice of using CNN's is evaluated by discussing existing literature and implementation challenges. Subsequently, the steps to building and simultaneous benchmarking the road symbol classification model and behavioural cloning model would be discussed in the third and fourth chapter respectively The fifth chapter will cover the drawbacks and other unexplored options pertaining to the current version of the models followed by the conclusion of the thesis.

Chapter 2: Methods

In order to develop the learning algorithm of the lane following model, the algorithm should be able to perform the following functions as a minimum:

1. **Feature Extraction**: The most important step towards developing a successful lane following algorithm is to be able to extract features from the input image. This can be achieved through several methods which can be based on edge detection, texture detection, luminance variance or a combination of all of these.
2. **Image Processing**: In order to make sense of the input data, the algorithm should be able to implement a series of filters and/or transformations to prepare the input image for feature extraction. Examples of image processing would include grayscale conversion, zoom and gaussian blur to name a few. It should be noted that several feature extraction techniques already comprise of image processing tools, hence the two functions might not necessarily account for two individual steps.
3. **Evaluation**: Post the application of the filters, transformations to the input image and the subsequent feature extraction, the model should be able to evaluate the image and make conclusive interpretations in order determine the path that the simulated vehicle must follow to showcase lane following abilities. For example: Determining the steering angle based upon the curve of the road.

There are a number of methodologies to attain lane following capabilities, however, as mentioned in section 1.3, only two were considered, namely Traditional Computer Vision and Deep Learning. Both these techniques are discussed in the subsequent sections.

2.1 Traditional Computer Vision

Traditional Computer Vision can be described as an Image processing tool that relies on finding obvious features from the image in order to perform classification or object detection. These features can be edges, corners, variance in luminance or even colour to name a few. In a general Traditional Computer Vision algorithm, the maximum number of features are extracted from the input images and these features form a type of each object class. During the evaluation phase, the algorithm looks for these set features in other images. If the algorithm is able to successfully recognise a set threshold of a feature type, it then classifies the image into an object type.

However, seldom is the case that to identify an object, only a single feature is utilised. Usually, it is a combination of feature detection that leads to object classification. As each feature usually requires a large number of filters and subsequently a plethora of parameters, it requires a large volume of work to fine tune the model and to get it working as per the desire. Apart from that, the traditional techniques rely on the judgement of the programmer to determine which features are required to be extracted in the first place. Hence implementing the traditional techniques calls for a substantial amount of trial and error from the programmer of the algorithm in order to get the model working. As it is crucial for the scope of this thesis that the model works on both structured as well as unstructured roads, it was important to take a deep dive into a number of these traditional techniques which are presented in the following table [5]:

Table 1: Road Detection Methodologies using Traditional Computer Vision Techniques[5]

Road Model Used	Feature Extraction Method	Post Processing	Remarks
Structured Roads	Texture Descriptors based features	None	Semi-Automatic, Not good with shadows, more descriptors selection is a scope of research
Structured Roads	Similarity Search (Region based features)	Knowledge based database is generated	Semi-automatic, efficient training is required
Structured roads with similar off-road regions	Edge detection based approach	None	Automatic
Structured roads with shadows	Illumination invariant image space & road model based approach	None	Better performance compared to HSI based colour space methods, suitable I in Shadowed environment
Unstructured Roads	Contextual Information , RGB descriptor based Bayesian framework for confidence map generation	None	Use of confidence map to increase the performance of SVM, KNN
Structured roads with lane marking	SUSAN Algorithm, Hough Transform (edge detection) adaptive Threshold	Inverse Projection, Tracking using Kalman Filter	Lane tracking and vehicle detection application
Structured and Unstructured roads	Vanishing point based boundary extraction and region extraction	Region based segmentation	36 Orientation with 5 scale at each pixels using Gabor Filters
Structured roads with lane marking	HSI colour model based feature extraction, Fuzzy C means algorithm based segmentation	Filtering after segmentation	Provides better results compared to RGB model based method for lane extraction
Structured road with lane markings	Colour based segmentation for lane extraction	Least square method for extension of lanes	Scope of research work in presence of light reflections, road signs
Structured roads with shadows	HSV colour space based method	Morphological post processing offline	Shadow Removal application

As can be observed from the table above, the technique of using Vanishing Point based boundary and region extraction showcases the most potential to be utilised as an all-purpose algorithm for both structured and unstructured road detection and subsequent lane following. The Vanishing Point Based algorithm is discussed in the next subsection.

2.1.1 Vanishing Point Based Boundary and Region Extraction

The Vanishing point road detection algorithm is a subset of Texture Extraction based models. Hui Kong et al. [6] in their paper address this road detection process by segmenting the detection process into two steps: the estimation of the vanishing point associated with the main (straight) part of the road, followed by the segmentation of the corresponding road area based on the detected vanishing point. The authors have implemented the method in an experiment containing 1003 generalised road images and have been able to develop a model that detects the road region even in various challenging conditions as shown below [6]:

Figure 3: Different types of roads where the Vanishing Point Road Detection is applicable [6]

The steps involved in the Vanishing point detection based method are as follows:

1. Confidence Weighted Texture Orientation Estimation
2. Locally Adaptive Soft Voting
3. Road Segmentation by dominant edge detection

The author in the papers [6][7] utilise the Gabor Filter for texture orientation estimation as they are known to be accurate. With the orientation φ and a scale ω, the Gabor wavelets are defined by [8]

$$\psi_{\omega,\emptyset}(x,y) = \frac{\omega}{\sqrt{2\pi}c} e^{-\frac{\omega^2(4a^2+b^2)}{8c^2}} \cdot \left(e^{ja\omega} - e^{\frac{-c^2}{2}}\right) \quad (1)$$

Where:
a = xcosφ + ysinφ
b = −xsinφ + ycosφ
c = 2.2

The above Gabor Kernel is used to implement texture orientation filter at each filter in the input image. A confidence map is then generated by using the formula below [6]:

$$Conf(z) = 1 - \frac{Average\,(r_5(z),\ldots,r_{15}(z))}{r_1(z)} \quad (2)$$

Where:
$r_1(z) > \ldots > r_{36}(z)$ are the ordered values of the Gabor response for 36 orientations.

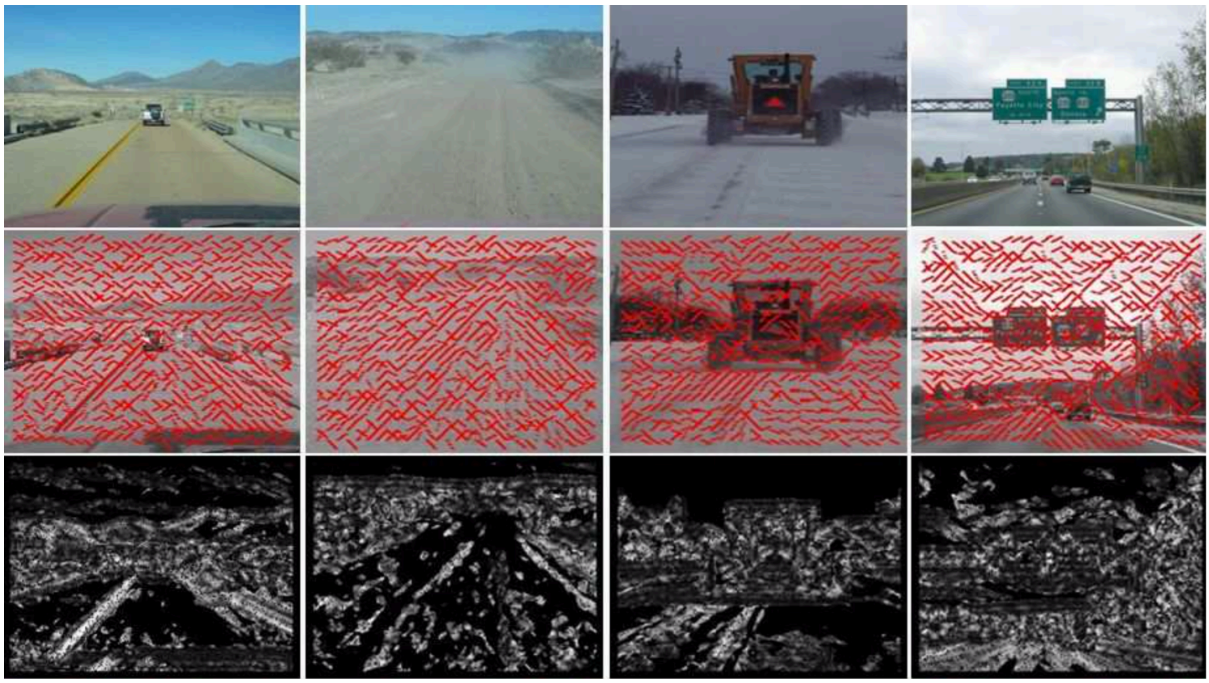


Figure 4: Gabor Filter Application [6]

Figure 4 above displays the application of Gabor Filter on a sample of four images as exhibited in the first row. The second row depicts the overlaying of the filters onto the sample images. The third row depicts the confidence maps computed using equation (2) for each pixel on the image. The brighter parts of the confidence map depict a higher value on the confidence scale of the orientation estimation.

Post estimation of the texture orientation through the confidence map based approach, Hui Kong et al. [6] offers an approach of Locally Adaptive Soft Voting. In this approach, each

of the pixels P with an associated texture orientation vector Q_p can vote for all pixels V in the subregion defined by R_v for a probable Vanishing Point Candidate. This region R_v is determined by the intersection of the Gabor response with a semi-circle below V, centred at V as shown in figure 5:

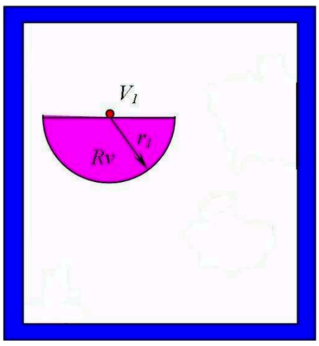


Figure 5: Localised Region for Selection of Vanishing Point Candidate [6]

After this initial estimation of the Vanishing Point by summarising the votes of every pixel, the next step is road segmentation. The objective of this step is to identify two of the most dominant edges, based on the initial vanishing point. This is then updated to find the exact, updated vanishing point.

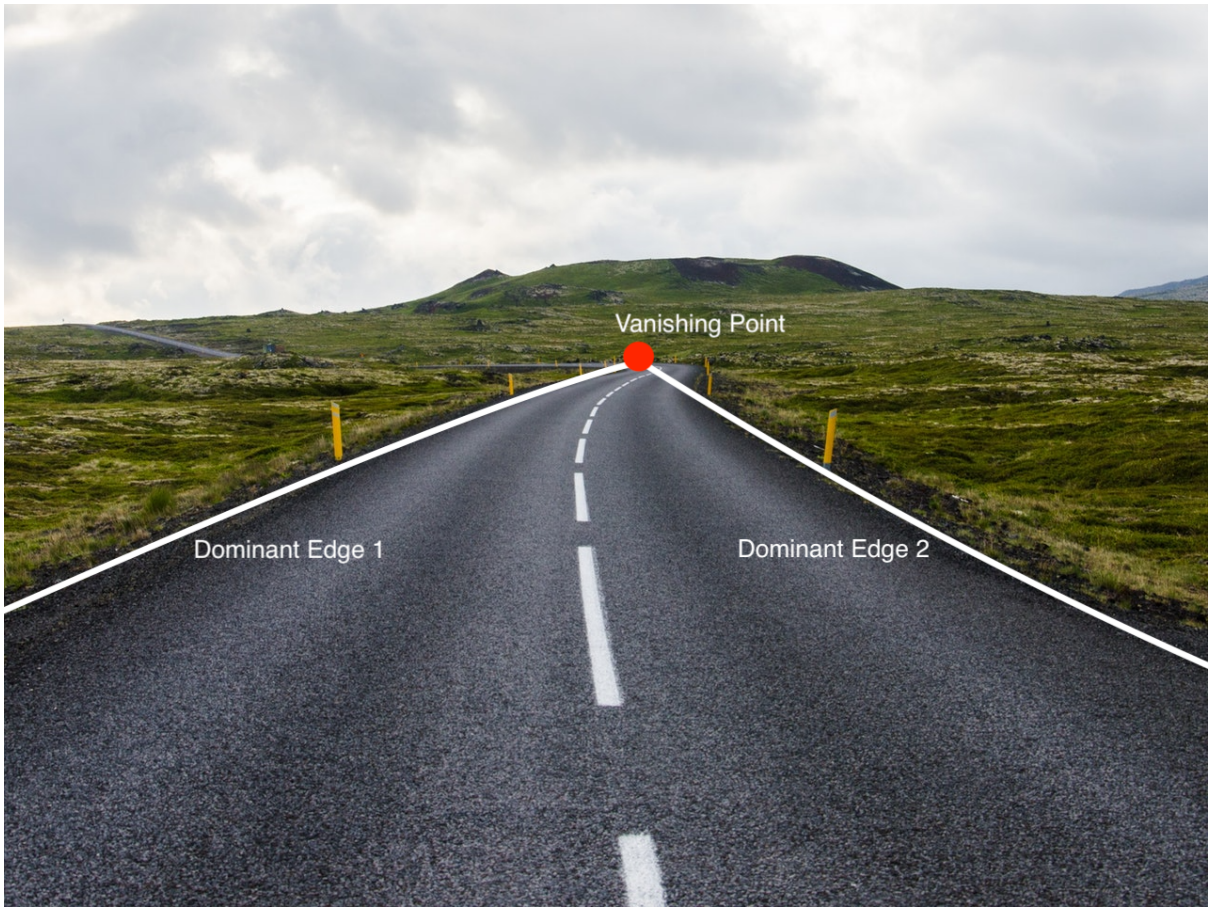


Figure 6: Road Segmentation Application

2.2 Machine Learning

Machine learning is the concept of building computational systems that learn over time based on experience at its most basic level rather than explicitly programming hard coded instructions into a system machine learning uses a variety of algorithms to describe and analyse data learn from it improve and predict useful outcomes. Often it is not enough to directly program a computer to perform a specific tasks such as driving a car. Tasks like Visual recognition and object detection are way too complex to just program in the form of hard coded instructions whereas the machine learning algorithm can learn and improve based on experience by interacting with its environment and then learning to detect and predict meaningful patterns to achieve desired results. The learning occurs between a learner and the environment which can be achieved through supervised or unsupervised learning.

Supervised learning is the most popular machine learning technique. Supervised learning typically begins with a training data set that is associated with labelled features (which define the meaning of the data) and find patterns that can be applied to an analytical process.

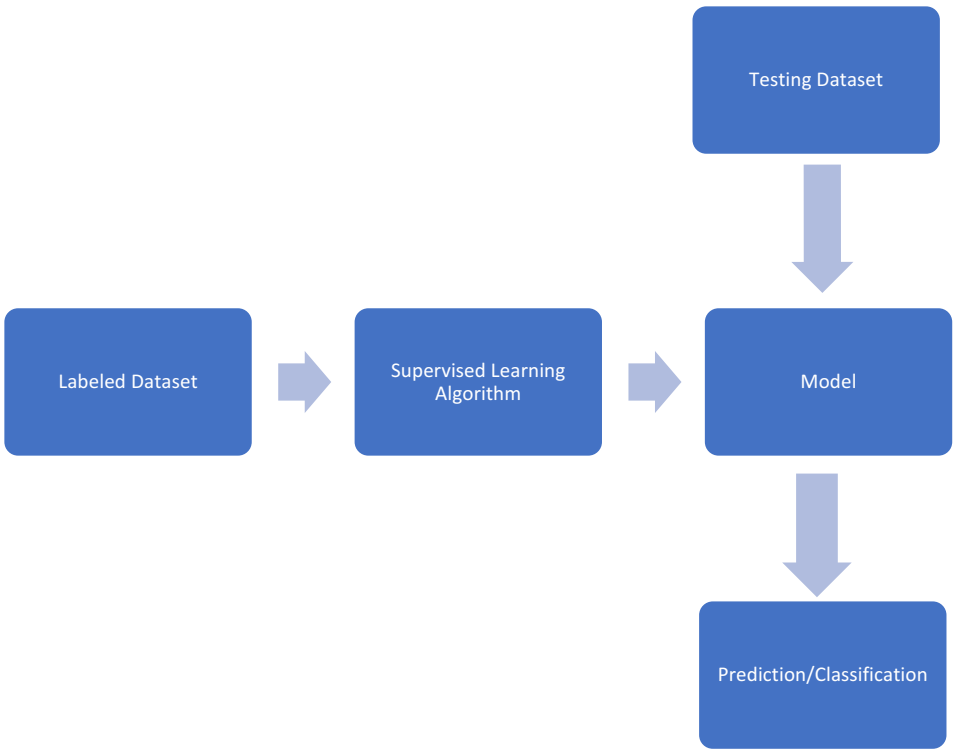


Figure 7: Supervised Learning Mechanism

As showcased in Figure 7, the supervised learning method is divided into three phases:

1. Training the Model
2. Validation
3. Testing the Model

In the training phase, a major portion of the training data set which comprises of labels for each data is fed into the model's algorithm. The algorithm then utilises this training data to

identify features, patterns and anomalies, to be then associated with corresponding labels. After this phase, the remaining portion of the training dataset is then fed into the model to validate the success rate. This step involves comparing the label of the validation data as predicted by the model to the actual label of the data. Upon receiving a high success rate, the model is then fed with new observations in the form of testing data set and is supposed to correctly classify the data into a label or make a predefined prediction according to the algorithm. The two major implementations of supervised learning are Classification, and Regression based models [9].

Another machine learning approach that occurs between the learner model and its environment is unsupervised learning. In this case, all the learner gets as training as large data sets with no labels and the learner's task is to process that data, find similarities and differences in the information provided, and act on that information without prior training. Two major uses of unsupervised learning are Clustering, and Association based models [9]. The following table highlights the categorical differences between the two:

Table 2: Difference between Supervised and Unsupervised Learning [9]

Parameter	**Supervised Learning**	**Unsupervised Learning**
Variables	For Training both input and output variables are provided in the form of data and associated labels	Only input variable in the form of unlabelled data is provided
Complexity	Simpler Method, however computational complexity depends on the implementation	More computationally taxing even in the simplest of implementations
Accuracy	Easier to obtain High Accuracy	Obtaining high accuracy is more challenging given the inherent complexity of this technique
Learning Platform	Learning is a separate offline stage and is not Realtime. However, prediction based on the learning can be Realtime	Learning takes place real time.
Used In	Classification, Regression etc.	Clustering, Association etc.

It should be noted that the generic accuracy of a machine learning algorithm is the number of correct predictions the model makes over the training data set.

Apart from these two methods, there also exists Semi Supervised and Reinforcement Learning techniques. However, for the scope of the thesis, the study was only limited to the former two techniques discussed above.

2.2.1 Deep Neural Network

Linear Model

Deep Neural Networks are made up of the fundamental building blocks called Perceptron. The functioning of the perceptron is based on the functioning of the human brain. It takes in input variables and processes these inputs by attaching respective weights and biases in the computational node which leads to a predictive score. The simplest form of a perceptron is a single cell Linear Neural Network which is showcased below:

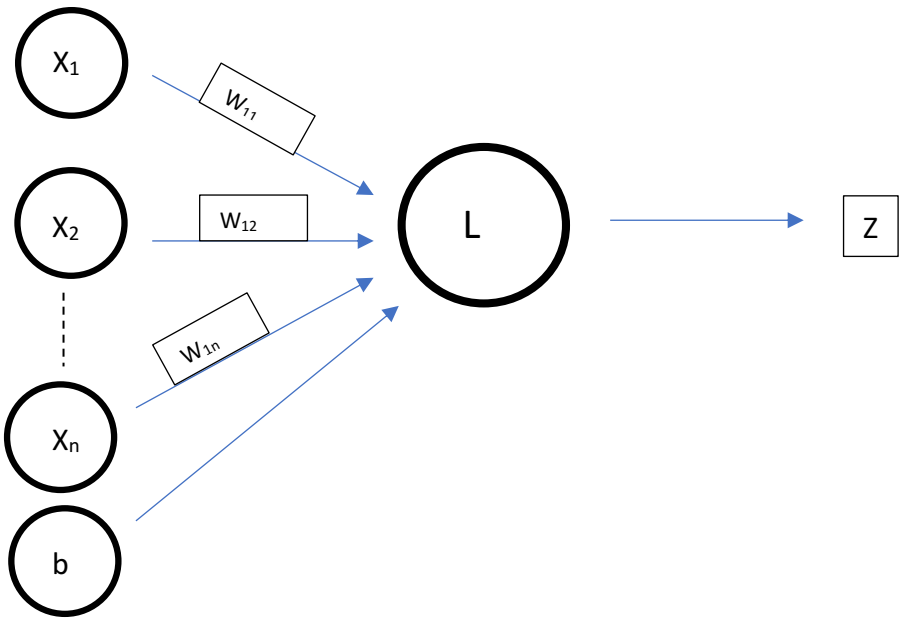


Figure 8: Linear Neural Network Model

The variables $X_1, X_2 \ldots X_n$ along with a bias 'b' form the input layer. They are associated with respective weights in the form of W_{1j}, where $j = 1,2,\ldots,n$ respectively. The node L represents the computational node. At this node, the sum of the signals from the input layer is computed and the result is then carried forward to either an activation function or another linear neural network in the form of an input, or even a combination of both. It is a combination of these Linear Models, that form a Deep Neural Network. Mathematically a Linear Model can be represented as follows [10]:

$$Z \longmapsto (w_{11}, w_{12}, \ldots, w_{1n}) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b \tag{3}$$

Or

$$Z \longmapsto w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n + b \tag{4}$$

Hence the Linear Neural Model can be identified as an affine map.

Model Training and Training Parameters

The objective of training a model is to identify the respective weights corresponding to each input. At the beginning of the training, each input variable is assigned a random weight and the model is made to predict the outcome. Each outcome corresponds to an overall error for the set of weights. The error is calculated by cross checking the predicted outcome with the labelled, predefined outcome in the training data. The model then re-adjusts the weights in such a way that the overall error is then reduced. This computation is performed until satisfactory levels of classification or regression is achieved.

In a broader classification, the model can be trained in one of two ways:

1. Pseudo-Inverse Solution
2. Gradient Descent Solution

The Pseudo-Inverse Solution is a methodology which involves calculations of pseudoinverse products [11] of each of these connected linear models in the form of a layer, and the target outputs. Tapson et al. [12] in their study conclude that this model is less computationally taxing compared to the Gradient Descent Method. However, setting up a model with a Pseudoinverse Training system is significantly more challenging than the Gradient Descent. Hence the scope of this thesis is limited to the Gradient Descent solution.

The Gradient Descent function works on the principle of generating a cost function. IN general, before a model outputs a prediction 'Z', the computed value from the node L is fed into an activation function. The purpose of this activation function is to generate a predictive score that can then correspond to a certain class, or can be utilised for regressive modelling. There are several types of activation functions such as Binary Step, Linear, Non Linear to name a few. However, in order to utilise the gradient descent, only non-linear continuous activation functions are a viable option, as the gradient for linear and binary step functions are a constant value. The activation functions will be discussed further later in this section.

The gradient descent algorithm can be described mathematically as follows:

1. All the weights, w_{ij} are assigned a small random value.
2. The cost function is calculated after the model computes its first iterative predictive outcome, represented as:

$$\Delta w_{ij} = \mu(t_i - y_i)m \tag{5}$$

Where
μ is the Learning rate
t_i is the target output of the specific node
y_i is the actual output of the specific node
m is the cost coefficient, which depends on the activation function

3. For each weight, the new values are identified by:

$$w_{ij} := w_{ij} + \Delta w_{ij} \tag{6}$$

The objective of these iterations is to create a sufficiently low cost function value such the gradient is nearly constant and the convergence is nearly zero. If implemented properly this will successfully achieve the desired classification or regression results.

The Learning rate is an important factor in this method as it determines the rate at which the gradient changes. A small learning rate, although might yield for much more accurate results, can result in the algorithm taking a long time to converge, or in case of a combination with a select few activation functions, cannot yield to convergence at all.

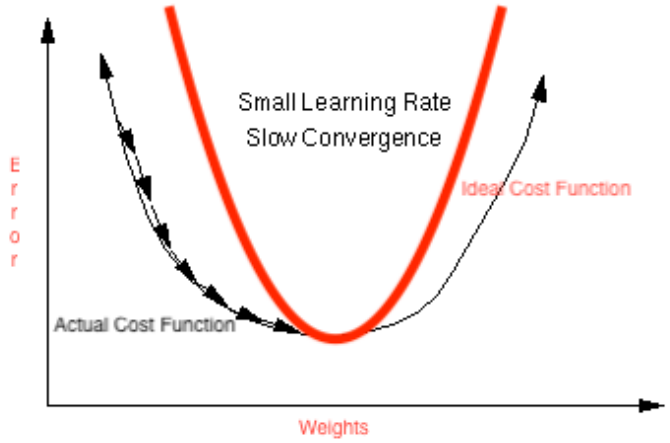


Figure 9: Small Learning Rate

On the other hand, a higher learning rate has very low accuracy and a high probability of divergence.

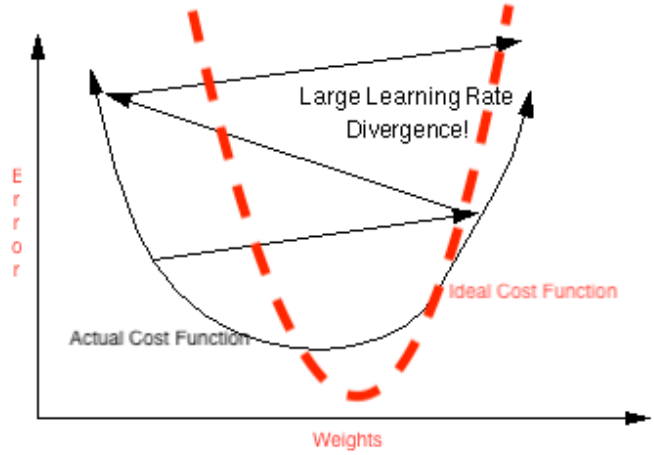


Figure 10: High Learning Rate

The Gradient Descent, as mentioned before is only as effective as the activation function it is used in combination with. The activation function is a tool that is used to convert the linear output of the computational node into a much more useful, prediction oriented output signal. In simpler terms, it is used to generate a predictive score from the output of the computational node which can then be utilised to predict the class/label or to implement regression. The need of the activation function increases as the complexity of the classification or regression model increases. As mentioned before there are three primary divisions of the activation function:

1. Binary Step Activation Function
2. Linear Activation Function
3. Non-Linear Activation Function

The Binary step functions work on a limit principle. The function generates binary outputs depending on whether the input to the function is higher or lower the threshold value. Hence this limits the functions to only binary classifications. It is represented mathematically and graphically as follows:

$$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geq 0 \end{cases} \tag{7}$$

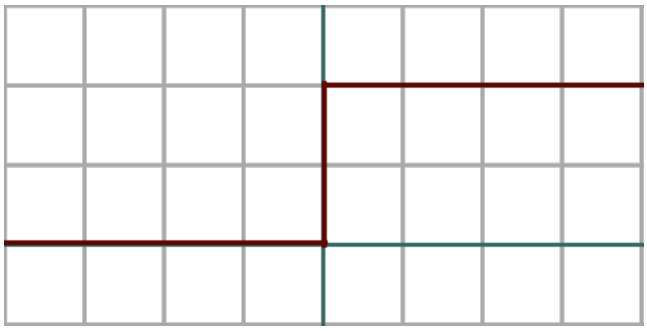


Figure 11: Binary Step Activation Function

The Linear activation function overcomes the limitation of the binary step but allowing multiclass classification. However, when used in Deep Neural Networks, the multiple interim layers of combinations of Linear neural network (called as Hidden layers), are rendered useless as the final layer will always be a linear function of the input layer. Hence eliminating the ability of the network to work with complex classifications by introducing severe redundancy. It is represented mathematically and graphically as follows:

$$f(x) = Ax \text{ for a constant } A \in \mathbb{R} \tag{8}$$

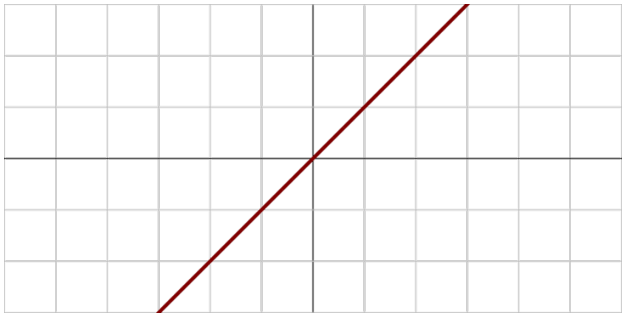


Figure 12: Linear Activation Function

Most of the modern Neural Networks utilise a combination of Non Linear Activation Functions to enable learning of complex classification and regression training data sets with a high accuracy rate as concluded by Serwa A [13] in his 2017 research article. The paper also showcases the importance of selecting the proper activation function according to the desired performance of the Neural Network.

There are a wide variety of Non Linear activations functions which can be utilised (possibly in combination with each other) to generate the desired output. C Nwankpa et al. [14] in his paper discusses the various types these Non Linear Functions and their utilisation trend in modern research. However, for the scope of this thesis, four out of the twenty-one functions

[14] are studied in detail purely based on the simplicity of their use and customisation, as well as debugging abilities. They are listed as follows:

1. **Sigmoid Function**: Jun Han et al. [15] defines the sigmoid functions as "a bounded differentiable real function that is defined for all real input values and that has a positive derivative everywhere." This property enables the prevention of abrupt "jump" in output values as in the case of linear and binary step function while still being bounded. It has been a popular tool for performing binary classifications or modelling logistic regression. However, this function has serious drawbacks in the form of vanishing gradient in deeper neural networks due to slow convergence, gradient saturation and non-zero centred output [14]. This function can be represented mathematically and graphically as follows:

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}} \tag{9}$$

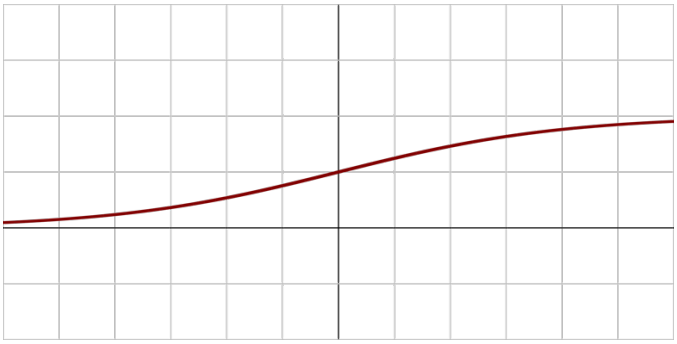


Figure 13: Sigmoid Function

2. **Hyperbolic Tangent (Tanh) Function**: This is a much smoother function and the range is bounded between [-1,1] as compared to [0,1] in the sigmoid function. It is also capable of enabling better learning when used in deeper neural networks as concluded by Radford Neal [16] and Bekir Karlik et al. [17]. The most common use of this function has been in the domain of Speech Recognition through Neural Networks. However, it is not able to solve the vanishing gradient drawback of the sigmoid as when the input is close to zero. This function can be represented mathematically and graphically as follows:

$$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})} \tag{10}$$

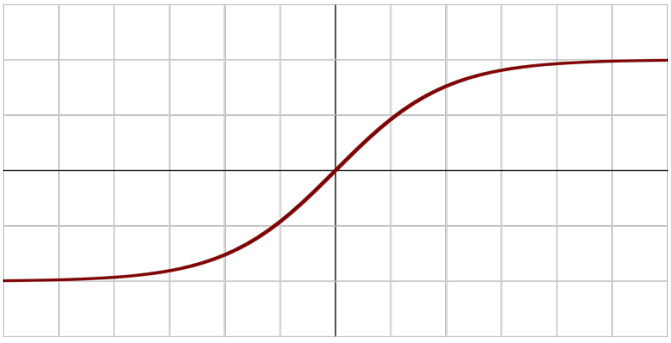


Figure 14: Hyperbolic Tangent Function

3. **SoftMax Function**: This function at its core is a probabilistic distribution tool. In the case of neural networks, this function returns the probability of output belonging to a certain class, with the sum of probabilities equating to 1. Hence this is used widely in Multi Class models and also in regression models in the output layers. The SoftMax function can be represented mathematically as follows:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{11}$$

4. **Rectified Linear Unit Function (ReLU)**: In 2010, Nair and Hinton [18] introduced the ReLU function, and ever since it has been the most powerful and popular activation tool to be utilised in the hidden layers of a Deep Neural Network. The headlining feature of this function is that it is a near linear function [14], and hence enables the easy optimisation features of linear activation functions and also combines the ability to work with gradient descent. This function also overcomes the vanishing gradient problem by forcing the negative input values to be converted to zero. Since this function does not compute any long divisions or exponentials, it also enables higher computational speeds as concluded by M. D. Zeiler et al. [19] in their article on the use of ReLU in speech processing and also by Aumit Leon [20] in his thesis about 'Accelerating Deep Neural Networks'. This function, however, is prone to overfitting and also can cause Dead nodes when the input is negative. The ReLU function can be expressed mathematically and graphically as follows:

$$f(x) = \max(0, x) = \begin{cases} x_i, if\ x_i \geq 0 \\ 0, if\ x_i < 0 \end{cases} \tag{12}$$

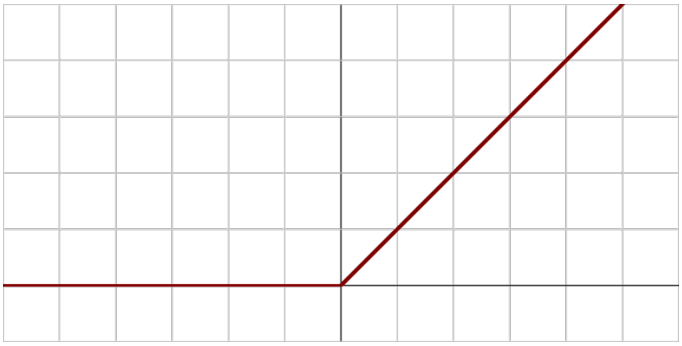


Figure 15: ReLU Function

2.2.2 Convolutional Neural Network

In the field of Image Analysis, the conventional Deep Neural Networks have a significant drawback in the form of providing average accuracy when dealing with, for instance, a complex classification. Images can be analysed as a matrix of numbers, with the resolution of the image bounding the size of the matrix, and the luminosity of each pixel determining the elements of the matrix. These elements range from 0-255 with the darkest regions assigned a value of 255 and the lightest a 0 in an eight-bit colour scheme. This can become extremely computationally taxing as the resolution of the image increases and also as the different colour channels (Red, Green and Blue) are introduced. For instance, take an example of an image of dimension 72 x 72 pixels. This image then corresponds to $72 \times 72 = 5184$ pixels. Now since the image is coloured, it also comprises of three colour channels (RGB), and hence the total pixel count of the image becomes $5184 \times 3 = 15552$. This value corresponds to the number of inputs to each node of a Deep Neural Network. In comparison, in an experiment was conducted to develop a Deep Neural Network model to accurately determine the handwritten digits from the MNIST Dataset which were provided by LeCun et al.[21], where the dimension of the single channel image was 28 x 28, hence only comprising of 784 pixels as input, the model displayed unsatisfactory levels of accuracy.

Figure 16: Illustration of a wrongly predicted digit by the Deep Neural Network

Hence the deeper the neural network is developed in order to tackle such three colour channelled images, the larger the complexity of computing the output. This hence exhibits the stark limitations of basic Deep Neural Networks in terms of Image Analysis. Hence it becomes inevitable to find a more efficient solution.

In recent times, the Convolutional Neural Networks (CNN) have been a dominant tool in the domain of analysis of visual imagery. They are a type of Deep Neural Network which utilise convolutional and pooling (also known as subsampling) layers to increase accuracy and computational efficiency when dealing with complex Image Analysis tasks. Although these convolutional and pooling layers can be arranged in multiple combinations to form different architecture, a typical CNN architecture can be showcased as follows:

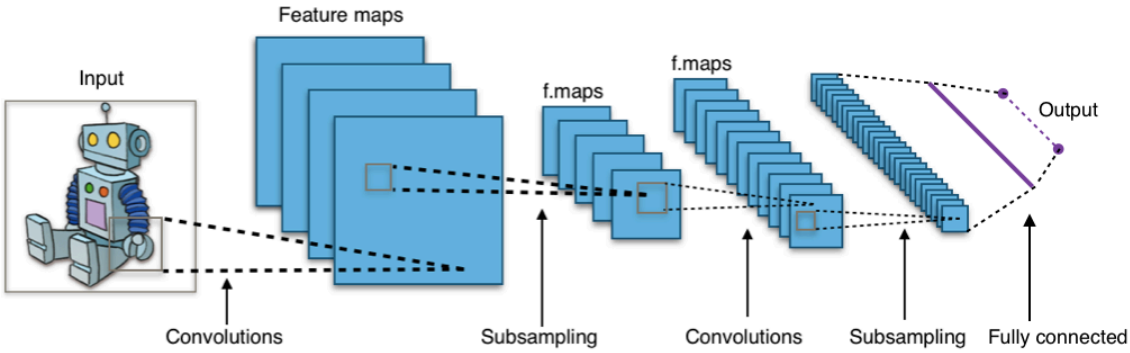


Figure 17: Typical Convolutional Neural Network [22]

As can be observed from the diagram above, the input in the form of an image is fed into a convoluted network of layers, which are then combined with pooling and other convoluted layers to be able to extract features and feed them into a Fully connected Deep Neural Network to obtain the classification or predictive score output. More details about convolutional and pooling layers will be discussed in the following sections.

Convolutional Layer

The convolutional layer has the task of extracting features from the images with the help of a kernel and developing a feature map comprising of layers of these features extracted through the kernel filter. The kernel filter is a small matrix of numbers, which is slid across or convolved across the width and height of the image matrix in a step by step move. It then calculates the dot product of the entries of the image matrix in each step with the entries of the kernel filter. As this is carried across all over the image, a two-dimensional array is developed comprising of the results from the dot product operations (numerical representation of a feature), which is termed as the feature or activation map of that specific kernel filter. These activation maps act as flags for the network to track the activation of a kernel when an unlabelled image is fed as input and hence determine the presence of the feature corresponding to the activated kernel. The features extracted by these kernels can be high level features such as edges or abstract features depending upon the depth of the network and the size of the kernel. The graphics below demonstrate the formation of a feature map through a kernel $K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ on an image matrix I:

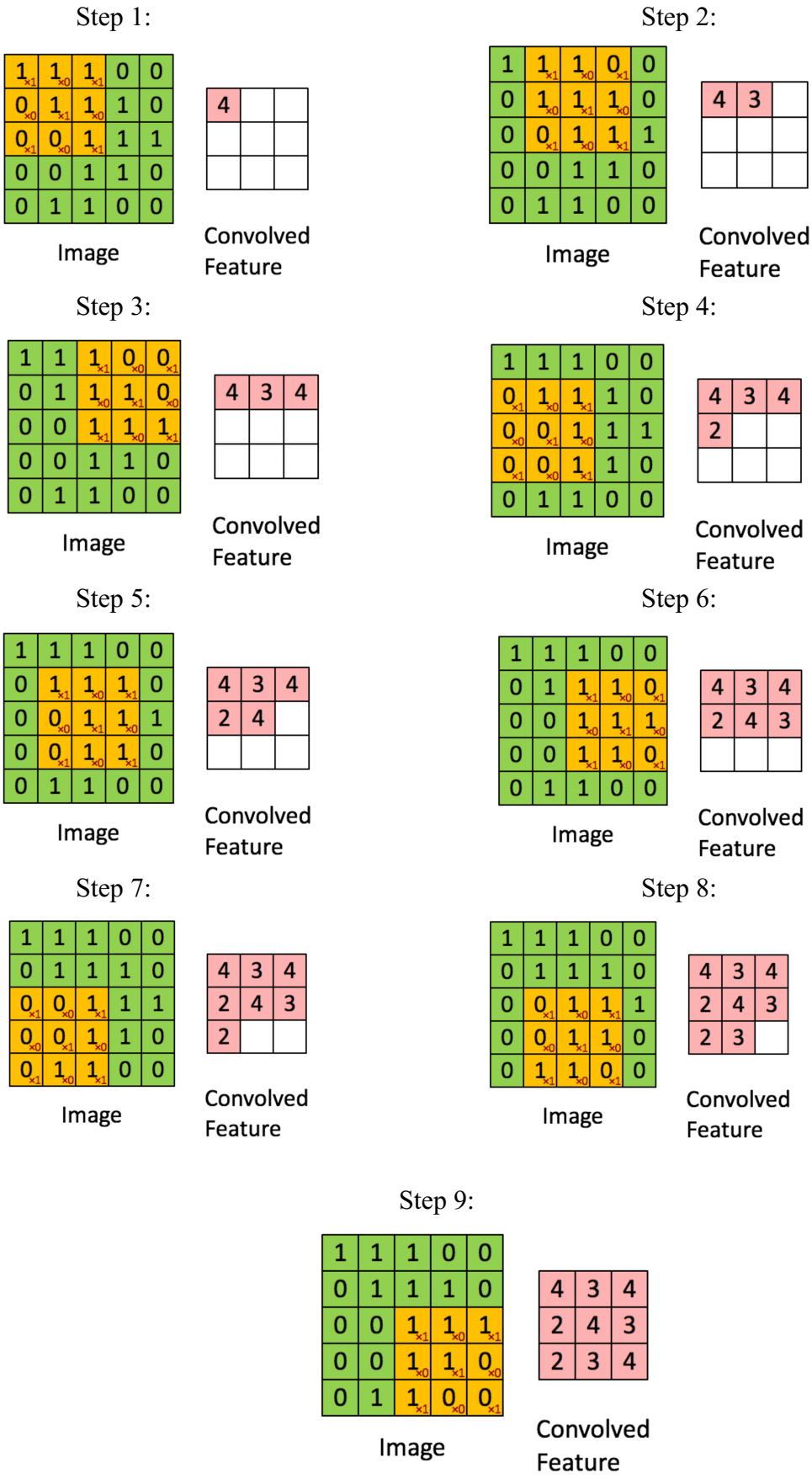


Figure 18: Illustrative steps of Feature Map formation through a kernel

Pooling Layer

In a general CNN architecture, the pooling layer is the interim layer between two convolutional layers or between the convolutional and the fully connected layer. The purpose of the pooling layer is to decrease the computation power required to process the data by reducing the dimensions of the feature maps. The Pooling layer focuses on extracting dominant features from the preceding feature map and omitting possibly noisy features, which can be the case in highly dense CNN's.

Similar to the Convolutional layer, a pooling kernel is slid across the width and height of the feature map in a similar step wise manner. In broader terms, the Pooling can be achieved either in the form of maxima's or in averages. This means that at each step, the kernel either outputs the maximum of the elements, or the average of the intersecting elements. This is illustrated in the following graphic where the kernel is a 2x2 matrix and the feature map is of the dimension 4x4, with the step size being 2:

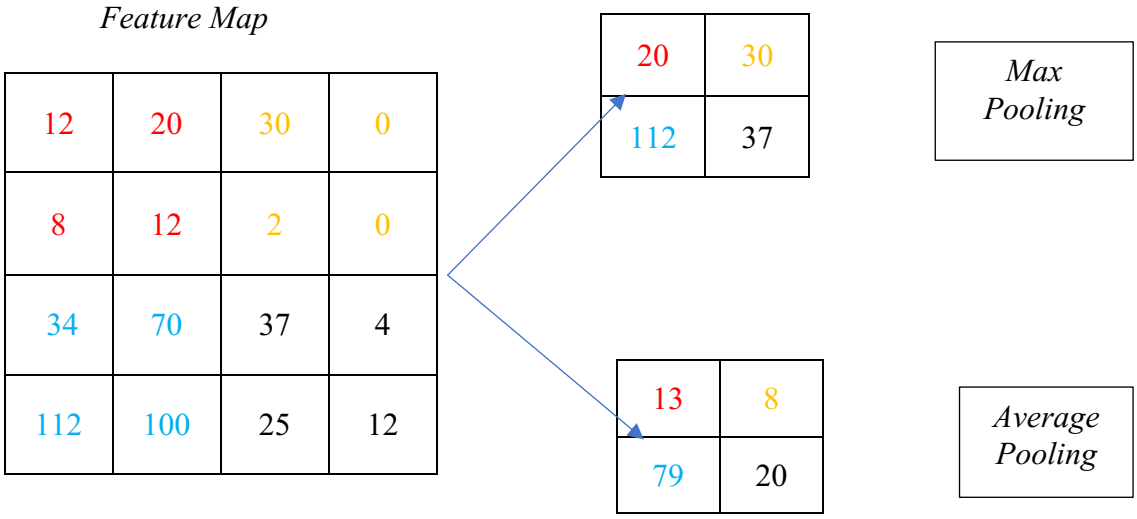


Figure 19: Types of Pooling

In the earlier implementations of CNN's, the average pooling method used to be a popular choice [23]. However recent trends in CNN models have shown an inclination towards the Max Pooling algorithm as showcased by M.D. Zeiler et al. [24] in 2014 and Xu et al. [25] in 2015.

There have also been attempts at eliminating the use the pooling layers altogether and utilising an all convoluted model without any significant compromise in the accuracy and efficiency of the model as exhibited by Spirengenberg Et. Al [26] in their article, where they critically examine the use of every component in a traditional CNN. However, this trend is beyond the scope of this thesis.

Fully Connected Layer

At its very core, the Fully Connected Layer (FC Layer) is an ordinary Deep Neural Network as discussed in the section 2.2.1. The FC layer is placed after the stacks of convolution and pooling layers. It takes the flattened output from the pooling layer as its input layer and computes the classification or predicted score using the activated functions, weights and biases. A basic illustration of the FC layer and its connection to the preceding stack of convolutional and pooling layers is showcased below:

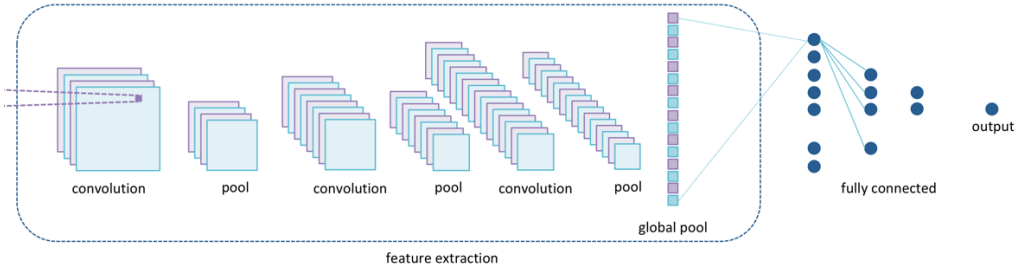


Figure 20: Illustration of connections between Convoluted and Pooling, and FC layer [27]

In this figure by M. Ntampaka et al. [27], it can be seen that the final pooling layer is flattened into a global pool layer which is then fed into the FC layer as an input.

Dropout

One of the challenges when dealing with deep CNNs is the amount of redundantly extracted features. These redundant features might complicate the process of classification or regression in the CNN. This can be caused by a variety of reasons such as lack of training data, exceptionally complicated features in the training data set, or even due to unwanted redundancy by overtly deepening the model. If not monitored, these may eventually lead to overfitting and hence can have significant negative consequences on the accuracy and efficiency of the model. One of the earliest proposals of mitigating overfitting was to utilise Data Augmentation by Simard et al. [28]. However, in 2014 Srivastava et al. [29] presented a far less computationally taxing tool for overcoming overfitting called Dropout.

Dropout attaches a probabilistic score to each of the nodes of the layer that it is applied to. The nodes with a threshold above, say, 'p' function normally, while the ones below the threshold are effectively turned into dead nodes. This not only reduces the computational load but also reduces redundancy and feature overloading, thereby countering the overfitting. It should be noted that Dropout is not exclusive to CNN's but can also be utilised by regular Deep Neural Networks with the same model of functioning as shown below:

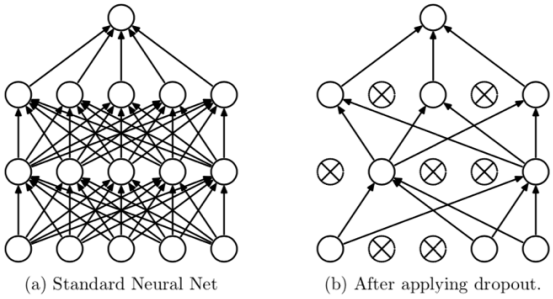


Figure 21: Illustrative use of Dropout [29]

2.3 Selection of a Methodology

The previous sections in the chapter introduced the various methodologies that can be utilised to implement a lane following model. In order to select a methodology suitable for the thesis it was important to assert on the following factors:

1. **Computational Resources:** One of the primary deciding factors was to find a balance between the computational power required by the model and the desired efficiency and accuracy of the model. As the computational resources available at the time of this thesis was only consumer grade, developing a complex model could have been a challenging task.
2. **Trends in the Image Analysis industry:** It was important to realise a forward trend upon which the model would be based. However absolute bleeding edge technology could have proved to be counterintuitive as the thesis was only limited to 24-30 weeks. Hence it was important to find a middle ground of deciding upon a methodology which had abundant existing literature yet was still being researched well. This would also be concurrent with the early objective of the thesis to make the model re-usable in the future.
3. **Optimisation capabilities**: The methodology selected was required to be customizable in order to obtain the desired results. Limitations on the utilisation of tools within an algorithm were considered as a liability.
4. **Convenient Implementation:** Due to the limitation on time and the computing power, a balance between convenient implementation and the performance of the model was to be taken into consideration.

Apart from these four, several other minor factors were taken into considerations while making decisions and choices during this study. These factors will be discussed in their respective sections in the following chapters. However, to decide the first, basic step of selecting the algorithm type for the model, a comparative table was developed. The table is a subjective representation of the advantages and disadvantages of each method corresponding to each of factors. The advantages and disadvantages are estimated, specifically based on this thesis and its scope. The table is represented as follows:

Table 3: Comparison Table

Factors	Traditional Computer Vision	Deep Neural Network	Convolutional Neural Network
Computational Resources	++	+	/
Trends	--	+	+++
Optimisation	-	/	++
Implementation	++	++	+
Score	1	4	6

The logic for assigning the score was relative. The context behind the scores were discussed in the previous sections of this chapter.

A method was awarded a '+' when it complied positively with the respective factors and the limitations associated with those factors. For instance, as the research on CNNs are extensively on the rise, even when compared to Deep Neural Networks, it was awarded with '+++' to not only show positive compliance but also indicate the gap between the amount of research currently underworks in the domains of CNNs and Deep Neural Networks. Similarly, '-' was assigned when using the method would hamper the compliance of the model with the respective factor. A '/' was awarded when the method neither contributed nor hampered with the compliance.

As the final step, the total aggregate score was calculated for each method, by assigning a value of 1 for each '+', -1 for each '-' and a '/' was assigned a score of 0. Hence it was decided to utilise CNNs as the basic framework for the desired Lane Following Model.

Chapter 3: Model for Classifying Road Symbols

3.1 Overview

The process of developing a Lane Following model was divided into two sections:

1. Develop a classification model using CNN to identify the road signals and accordingly determine the driving commands of the autonomous car.
2. Develop a continuous regression based model using CNN to be successfully able to clone the behaviour of a manually driven car.

In this chapter, the process of developing the Classification model is discussed. The programming language utilized was Python 3.6, with libraries and packages obtained from the open source Anaconda distribution. The text editor utilized was Atom, developed by GitHub. However, during the later stages of the study, in order to utilise GPU prowess, Google Colaboratory tool [44] was used. Some of the python libraries utilized in developing both the models are listed as follows:

1. NumPy
2. MatPlotLib
3. Keras
4. Random
5. Pickle
6. Pandas
7. Cv2
8. Os
9. NTpath
10. imgaug

It is worth noting that in the earliest stages of the thesis, it was planned to utilize MATLAB as the main programming tool as the model was initially supposed to be based on traditional computer vision algorithms, specifically the vanishing point algorithm (Section 2.1.1). However, after discovering the Convolutional Neural Networks, and establishing it as the central framework for classification and regression modelling, Python presented a significantly better environment in terms of intuitiveness and abundance of literature as concluded by Taylor et al. [30] in their comparative study between MATLAB, Python and R over a range of applications.

3.2 LeNET Architecture

The LeNET architecture [31] is one of the earliest architectures of a CNN. LeNET along with the MNIST [21] architecture can be termed as the 'Hello World' equivalent of the CNN domain. It consits of seven layers, excluding the input layer. All the layers are associated with weighted nodes. The size of the input image is 32 x 32. This input is connected to a stack of two convoluted and two pooling layers along with the FC layer to generate an output. The LeNET model also has the ability to run solely on the CPU, without the need for a powerful GPU for simpler classification tasks. The standard LeNET model can be represented graphically as follows:

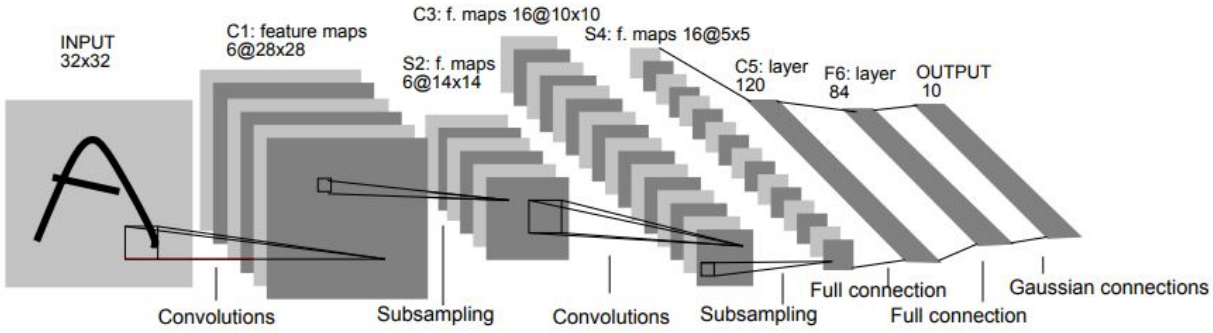


Figure 22: LeNET Architecture [31]

As the model to be developed has a relatively uncomplicated task of identifying and classifying road symbols, and given the conditions and challenges in Section 2.3, the LeNET had potential to be used as the basic framework.

3.3 Dataset

The dataset was curated by Jad Slim [32] on a Git Repository. It comprises of German Traffic Signs which are divided into 43 classes. The dataset is divided into training, validation and testing data in the form of a pickle file and a labelled dataset titled 'signnames', which stores the labels for the corresponding signs. The properties of the dataset are tabulated below:

Table 4: Dataset properties

	Training Data	**Validation Data**	**Testing Data**
No. of Images	34799	4410	12630
Dimension of each image	32 x 32	32 x 32	32 x 32
No. of color channels	3	3	3

The following is the distribution of the number of images in each of the classes in the chosen Dataset:

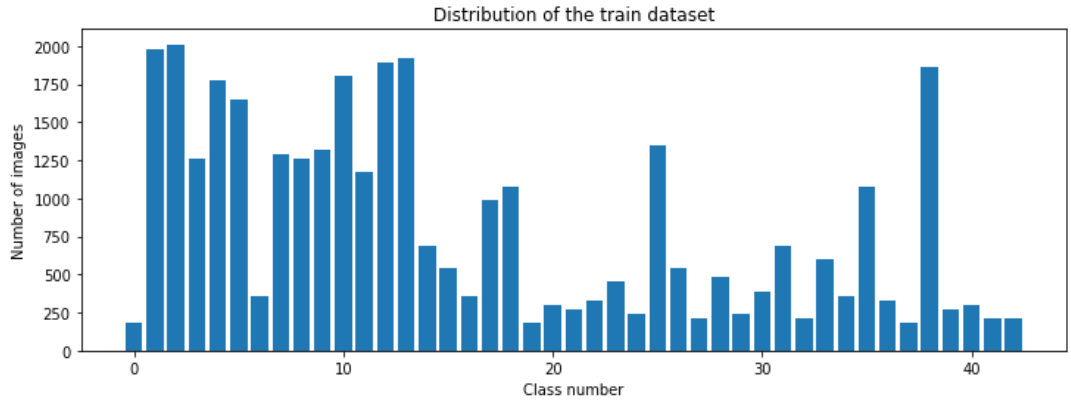


Figure 23: Dataset Distribution

3.4 Pre-processing Images

A complex dataset can have images belonging to a particular class, but at different perspectives, angles or with different luminosity levels (such as images from daytime and night time). This non uniformity can trick the model into overfitting. Hence, in order to utilise images from complex and serialised datasets like the German Traffic sign dataset, it is important to pre-process the data in order to generalise the data to assure better learning of the model. Generalisation introduces uniformity in the dataset. The generalisation is divided into the following steps:

1. **Grayscale Conversion:** The purpose of this step is to reduce the taxing computation by converting the three channel images into a grayscale, single channel image. Sermanet et. Al [35] concluded that colour does not improve the performance of the model significantly hence converting to grayscale gets justified. The following is the code snippet [62] and the subsequent grayscale conversion from a sample image

```
def equalize(img):
    img = cv2.equalizeHist(img)
    return img

img = grayscale(X_train[1000])
```

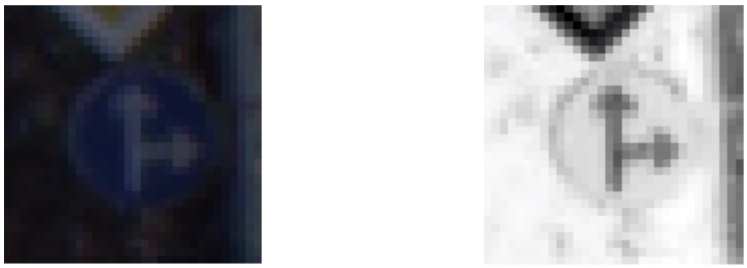


Figure 24: Grayscale Conversion

2. **Histogram Equalization**: This method is defined by Singh et al. [33] as "a spatial domain method that produces output image with uniform distribution of pixel intensity means that the histogram of the output image is flattened and extended systematically." This is done in order to counteract the different luminance levels of pixels in images belonging to the same class and, hence enabling uniformity. The following is the code snippet [62] for histogram equalization, along with a graphic illustration by Singh et al.

```
def equalize(img):
    img = cv2.equalizeHist(img)
    return img

img = equalize(img)
```

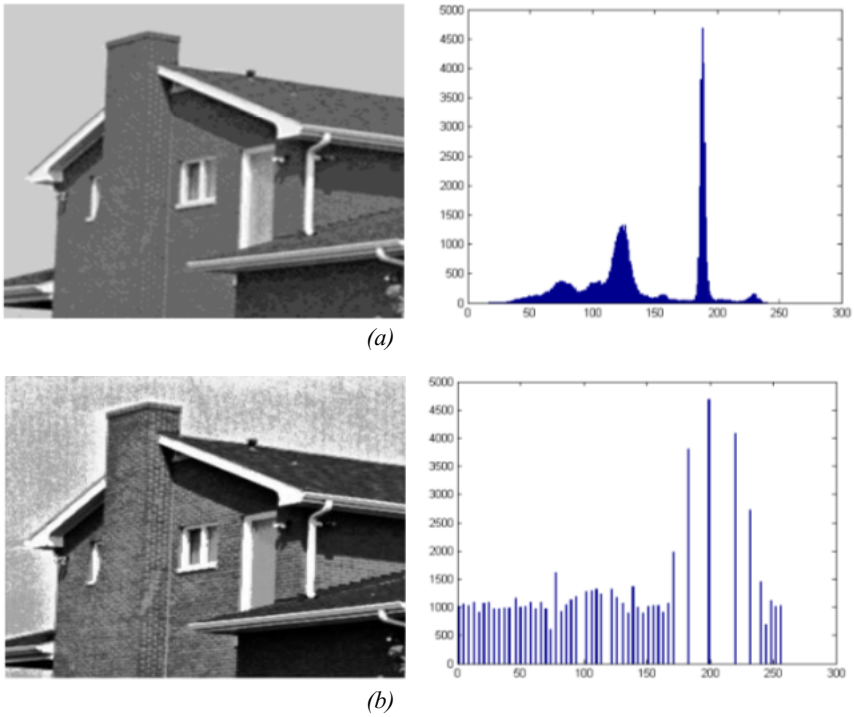


Figure 25: (a) exhibits the pre equilazation and (b) exhibits post equalization [33]

The difference in pixel intensity distribution is fairly evident from the figure above as the curve seems to have flattened out significantly. It is to be noted that the equalization only works with a grayscale image [33].

3. **Normalization:** In this process, the pixel intensities which range from 0-255 are normalized between 0-1 to reduce the computational load. This is achieved by simply dividing each of the pixel density by 255.

Although these pre-processing techniques significantly improve the learning ability of a model, depending upon the complexity of the data set and also the size of the training data set available, these techniques as a stand-alone may not be sufficient to get accurate results. Hence data augmentation techniques can be used in such scenarios. They will be discussed in detail in future sections of this chapter.

3.5 LeNET Implementation

After pre-processing the image dataset, the next step was to implement the LeNET architecture based model. As discussed in section 3.2, the Road Classification Model will also have seven layers, along with a dropout layer to prevent overfitting. The kernel size for the first convolutional layer is 5x5 and for the second convolutional layer is 3x3. The kernel size for both the pooling layers is 2x2. Other properties are tabulated below:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 28, 28, 30)        780
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 30)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 12, 12, 15)        4065
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 6, 6, 15)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 540)               0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               270500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_4 (Dense)              (None, 43)                21543
=================================================================
Total params: 296,888
Trainable params: 296,888
Non-trainable params: 0
```

The learning variables for the model training were set as follows:

1. Batch Size = 400
2. Epoch = 10
3. Shuffle = 1
4. Verbose = 1

The batch size is the number of samples processed before the model is updated. The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset. The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset. The Shuffle is used to shuffle the entire dataset and make the batches accordingly. Verbose is used to indicate the progress of the training through each epoch.

The model took about 290 seconds to train and itself with a learning rate of 0.01. However, the accuracy obtained through the Adam optimiser [34] for the training data was 96.24%, but for the validation data, it was a mere 92.63% as shown in the graph below:

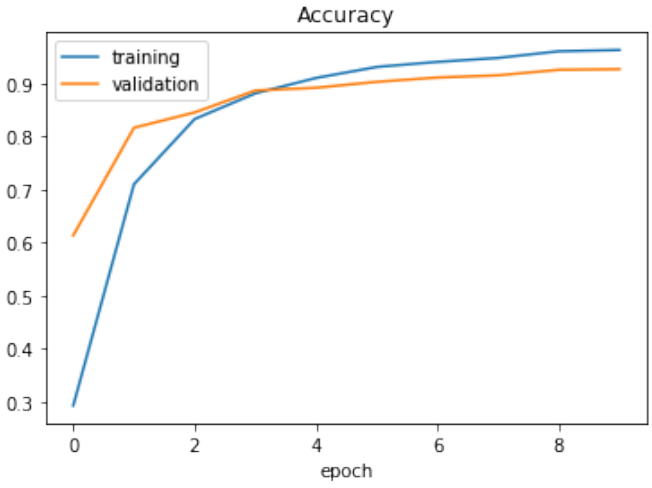


Figure 26: Accuracy results for the LeNET implemented model

The model exhibits that the validation set was not able to match the accuracy of the training set through the entire run of training the model as the training accuracy surpassed the validation accuracy between the third and fourth epoch. This is hence a case of overfitting. An even lower accuracy rate of only 91.1% was achieved when the model was evaluated with the testing data set. Hence it was necessary to make changes in the current implementation of LeNET model.

3.6 Fine – Tuning Model

The objective of fine tuning the model was to tackle two major issues:

1. Improving the accuracies of both training and validation dataset learning from their current below par values (about 96 % and 92% respectively) to a good or excellent level of about 98.0-99.5%.
2. Treating the Overfitting problem.

In order to improve the accuracy, a good approach would be to increase the number of features extracted by the convolution kernels, by increasing the number of filters, thereby increasing the depth of the feature map. Hence the depth of the first convolutional layer was changed from 30 to 60, and in the second convolutional layer, it was changed from 15 to 30. Subsequently it also changed the shape of the pooling layers connected to the respective convolutional layers to 60 and 30 in depth. Doubling the size of the feature map nearly doubled the number of training parameters in the computation to 579,833.

This yielded in a significant increase in the training accuracy to 98.09%, however, the validation accuracy was still a paltry 93.51% as can be seen from the figure below:

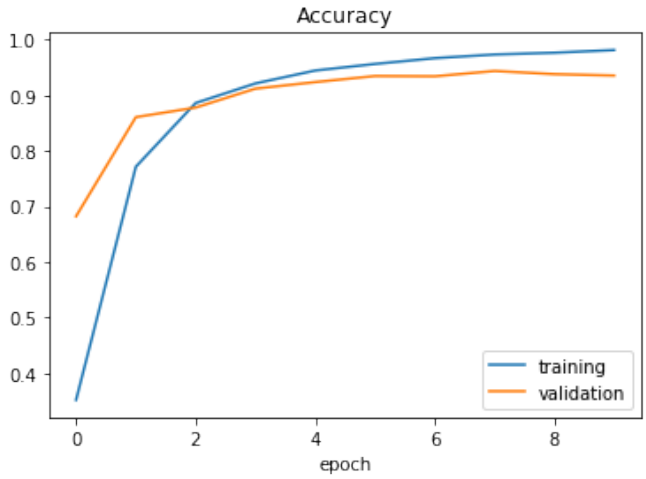


Figure 27: Accuracy result after first fine tuning attempt

It is to be noted that the accuracy obtained by evaluating the model with the training data set also increased from 91.1% to 92.79%, however overfitting is still present.

In the second attempt to improve the accuracies, the trend of extracting more features from the training dataset was persisted with. However, in this attempt instead of increasing the depth of the feature maps, two additional convolutional layers were added. This method

enables extracting more complex features which potentially could lead to higher accuracies. Hence the new structure of the model was as follows:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 28, 28, 60)        1560
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 24, 24, 60)        90060
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 60)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 10, 10, 30)        16230
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 30)          8130
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 30)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 480)               0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               240500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_4 (Dense)              (None, 43)                21543
=================================================================
Total params: 378,023
Trainable params: 378,023
Non-trainable params: 0
```

It is to be noted that the number of parameters has dropped to 378,023 from the previous value of 579,833. The reason for this is that with each convolutional layer, the size of the input decreases due to the kernel filtering. However, this decrease in the number of trainable parameters need not necessarily yield a faster computational time, as the complexity of the model has increased with the additional layers.

This attempt bettered the previous accuracy results by increasing the training efficiency to 98.94% and the validation efficiency to 97.03% as showcased in figure 28. The evaluated accuracy from the training dataset also increased to 94.76%.

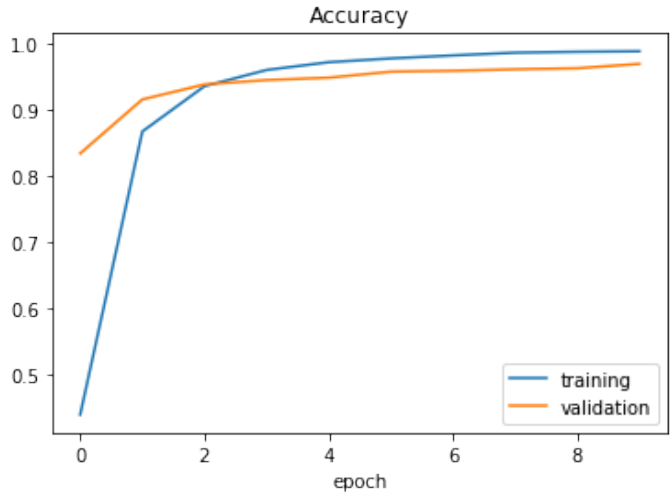


Figure 28: Accuracy results after second fine tuning attempt

Now that the training accuracy was suitably high, the objective was to treat the overfitting of the model, possibly increasing the validation accuracy in the process. Hence as mentioned in the previous chapter, section 2.2.2, the dropout tool could potentially be utilised again. Therefore, a dropout layer was placed immediately after the second pooling layer. The theory behind this placement was to retain the complex feature extraction properties of the model by not tinkering with the convolutional and pooling layers. The results for this attempt were as follows:

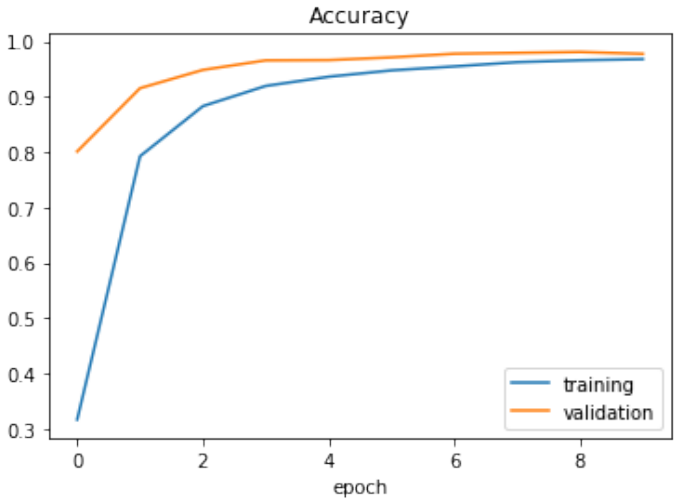


Figure 29: Accuracy results after third fine tuning attempt

Although this attempt did solve the overfitting problem of the model, it did not deliver expected results. The additional dropout layer decreased the training efficiency to 96.9% and did not significantly boost the validation accuracy as well as it changed from 97.03% to 97.76%. This result is understandable. Because the dropout layer was placed after the feature extraction segment of the model, by switching off half of the nodes that delivered information about the extracted features to the fully connected layer, it hampered the learning abilities of the model. Hence another attempt was made but instead of switching off half the nodes, in this attempt, only 30% was disabled, by changing the dropout value from 0.5 to 0.3. The following were the results:

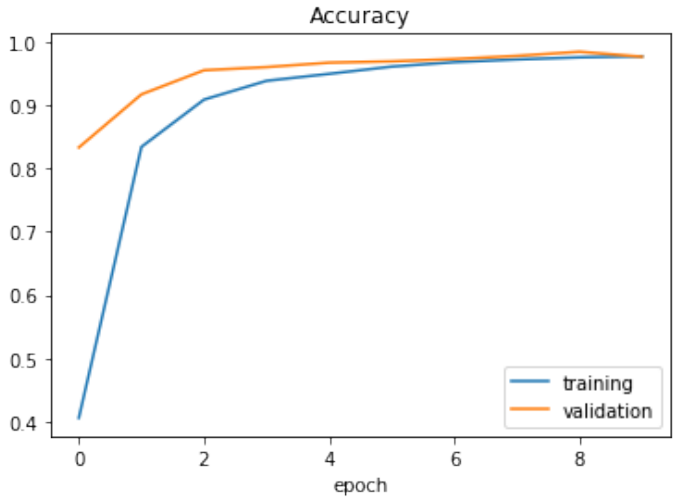


Figure 30: Accuracy results after fourth fine tuning attempt

This attempt did improve the results. The training accuracy reached 97.98% and the validation accuracy was nearly the same at 97.76%. It should also be noted that the evaluative accuracy from the training dataset increased to 95.83%. This result is closer to the desired result of achieving a 98% or higher accuracy, yet it seems that there remains room for improvement.

In this final attempt, the technique of Data Augmentation was explored to prevent overfitting whilst increasing the efficiency. The technique was not implemented previously because it required significant graphic computing prowess. Hence in order to make this attempt feasible, a Graphics Processing Unit was added to the setup.

With Data augmentation the main objective to increase the amount of data fed into the model for training whilst treating data imbalances between various classes as exhibited in figure 23. In a survey paper by Connor Shorten et al. [36], the traditional data augmentation techniques proved to have appreciable performance improvements. These traditional data augmentation techniques are comprised of the following modifications:

1. Width Shift
2. Height Shift
3. Zoom
4. Shear
5. Rotation

Although Mikołajczyk et al. [37] concluded that several modern techniques such as texture transfer and style transfer when combined with traditional techniques can further improve the efficiency, considering that nearly 98% accuracy was already achieved even without the use of data augmentation, the study of this thesis was hence limited to traditional techniques itself.

The Keras library [43] provides an inbuilt Data Augmentation function called 'ImageDataGenerator'. Using this function all the images from the dataset were given the following modifications:

1. 10% Pixel shift across the width
2. 10% Pixel shift along the height
3. 20% inner zoom or crop
4. 10% Shear
5. 10 degrees of Rotation

The first two modifications help balance out off-centred images. The third modification was to counteract the cropped images variance. Shear helps with the images that have been shifted in perspective. Rotation finally balances out tilted images. All these modifications, as mentioned before, are done to generate uniformity in the dataset. These can be implemented by the following snippet of code [62]:

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(width_shift_range=0.1,
                             height_shift_range=0.1,
                             zoom_range=0.2,
                             shear_range=0.1,
                             rotation_range=10.)

datagen.fit(X_train)
```

It is to be noted that the learning variables need to be modified when using data generation. The batch size is now decreased to 100. The learning rate is also modified to 0.001. A new variable called 'steps_per_epoch' is introduced. It determines the size of the augmented data. For instance, if we need to generate 100,000 augmented images, with a batch size of 100, the value of steps_per_epoch should be 1000. Hence for the current model, steps_per_epoch was set to 2000 in order to generate 200,000 augmented data. This is nearly 6 times the size of the earlier training dataset with size 34,799. The dropout layer added in the fourth attempt was also removed from the model.

The results after implementing all these modifications were as follows:

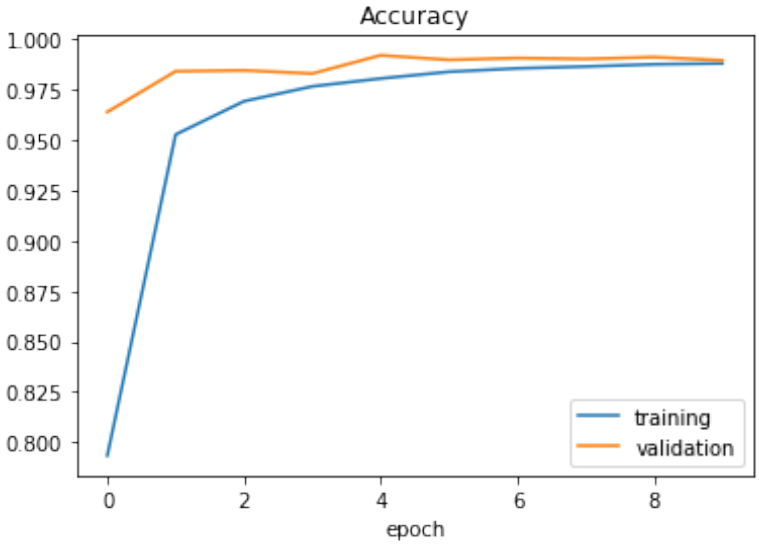


Figure 31: Accuracy results after fifth fine tuning attempt

Clearly, the data augmentation technique was a success. The training accuracy now stands at 98.92% and the validation accuracy now stands at 99.12%. It should also be noted that the evaluative accuracy with the training dataset now stands at 98.01%.

Chapter 4: Model for Mimicking Human Driving Pattern using Behavioural Cloning

4.1 Overview and Model Workflow

The model was developed in order to emulate human driving behaviour in an autonomous car. This, as mentioned in section 1.1, theoretically has the potential to reduce the limitations on the level of autonomy in a vehicle. As the model would be capable of extracting a plethora of features connected through layers of convolutions, its dependency on a single marquee feature, such as lane markings on a road should be negligible, when it comes to guiding the car through the tracks. It would hence be critical to produce a well-trained, generalized model which emulates the driving pattern of a human driver, within acceptable limits of error. The acceptable limits are defined by the error values that do not lead to crashing or anomalies in the behaviour of the autonomously run vehicle. In the case of this study, the errors and subsequently derived parameters would be based on the results obtained through the validation data set.

The following is the workflow of the model during the learning phase:

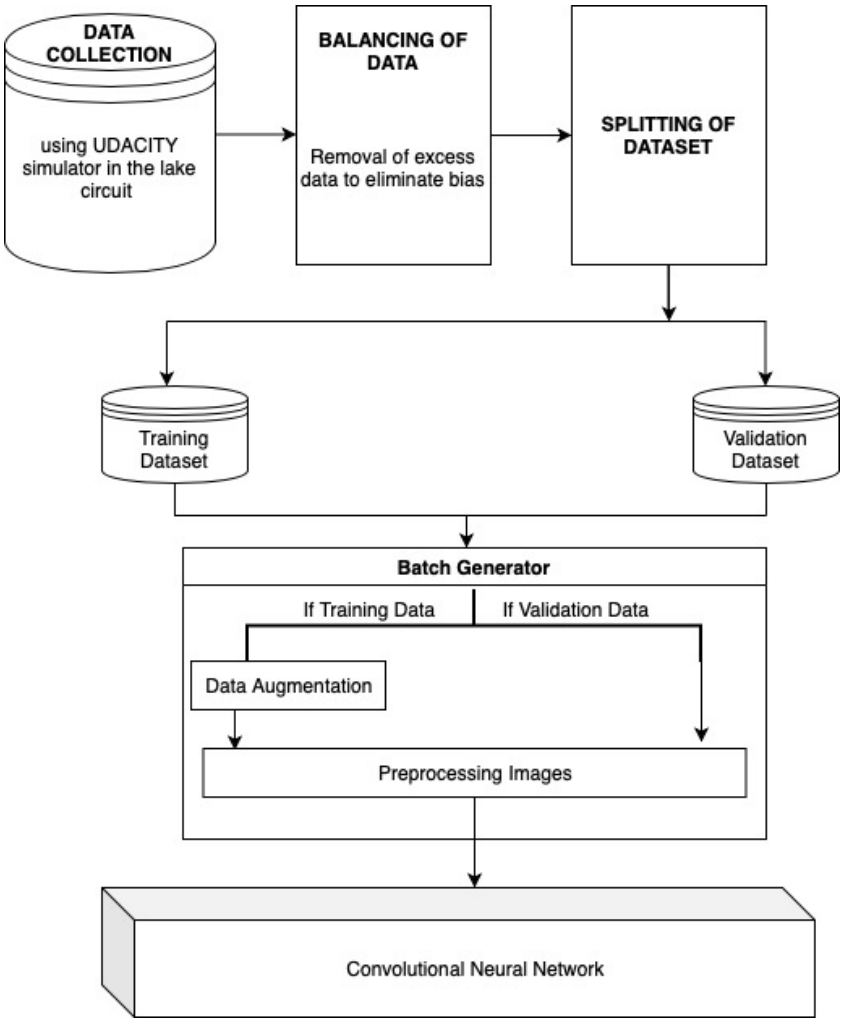


Figure 32: Model Workflow during the learning phase

4.1.1 Polynomial Regression

Polynomial Regression is used to predict continuous values across a continuous spectrum of data. In statistics, polynomial regression is a form of linear regression in which the relationship between the independent variable and the dependent variable is modelled as a polynomial [38]. The objective of this regression based model to suitably fit the non-linear training data set with minimal loss, that is determined by the mean squared estimate as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \quad (13)$$

The model is then curve fitted using this MSE loss function by adjusting the weights between the nodes in order to minimise the loss function. However, instead of outputting multiple class labels as in the case of the Road Symbol Classification model in the previous chapter, the model outputs a predictive output value using the fitted curve in the training phase. An illustration of curve fitting through a polynomial regression model is shown below:

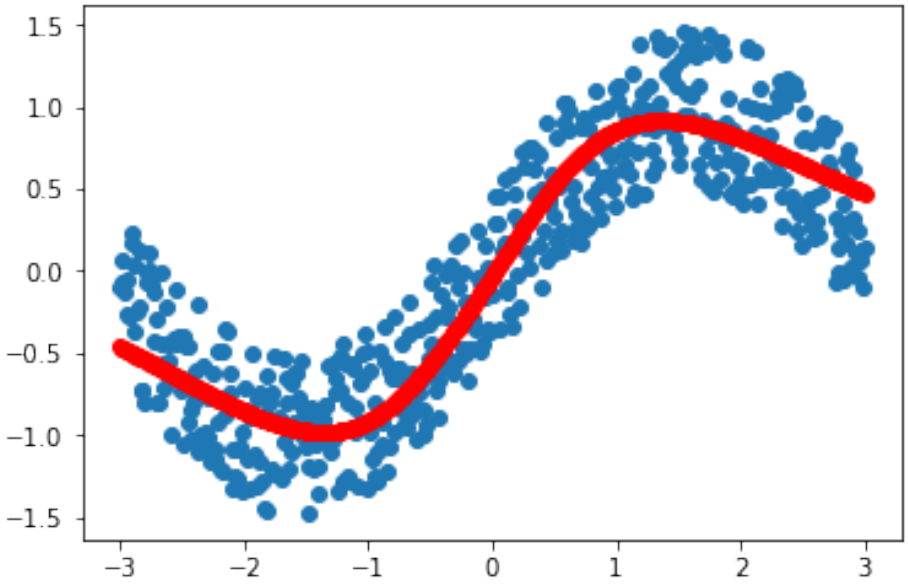


Figure 33: Illustration for Polynomial Regression curve fitting

It is to be noted that the blue coloured points on the graph are analogous to the training data and the red curve is analogous to the polynomial regression based prediction curve for the model.

4.1.2 Nvidia Architecture

The Nvidia Architecture introduced in April of 2016, by Bojarski et al. [39], forms the backbone of this thesis study. The architecture featured an autonomous vehicle model with end to end learning capabilities. The model is capable of mapping image inputs into direct steering commands. The model is also capable of operating in areas with no visual guidance or markings (such as lane marking) such as on unpaved roads or parking lots. This was possible because the model was never trained explicitly to extract lane marking or other visual guidance features. This hence made this architecture an ideal base for this thesis.

The method for training the model is depicted by Bojarski et al. [39] as follows:

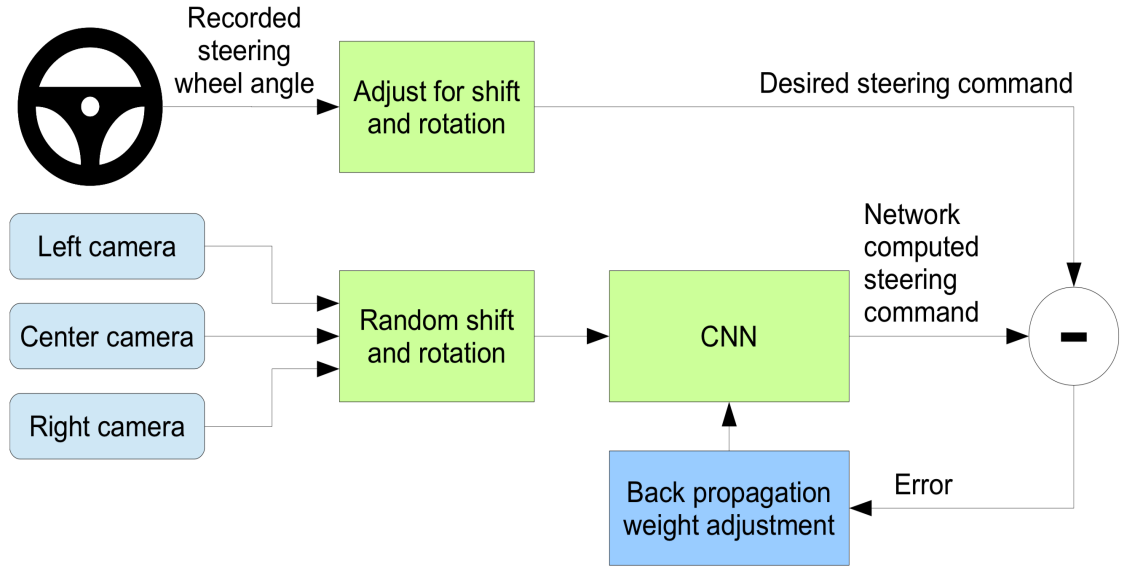


Figure 34: NVIDIA model training

As can be observed, the CNN works on the error minimisation method. The CNN is allowed to make predicted steering angles from the labelled dataset comprising of the images captured from the left, right and centre cameras mounted on the vehicle. All of these images are labelled with the actual, desired steering angle (along with road type, weather condition and driver's activity). The predicted steering angle is compared to the actual steering angle, and through the error associated with this comparison, the weights are adjusted, and the model is accordingly fitted. The presence of the data augmentation technique in the form of a random shift and rotation block from the diagram should be noted.

The CNN comprises of five convolutional layers along with 3 fully connected layers. The network also consists of a normalization layer. The normalization is used to increase the convergence speed of the error metric and also reduces co-dependence between the layers, which can potentially lead to overfitting as stated by Fabian Schilling [40] in his thesis. For the first three convolutional layers, the kernel size is 5x5 with a stride of 2x2, and for the final two layers, the kernel size is reduced to 3x3 with a regular stride of 1x1. The architecture is graphically depicted in the following figure by Bojarski et al. [39]:

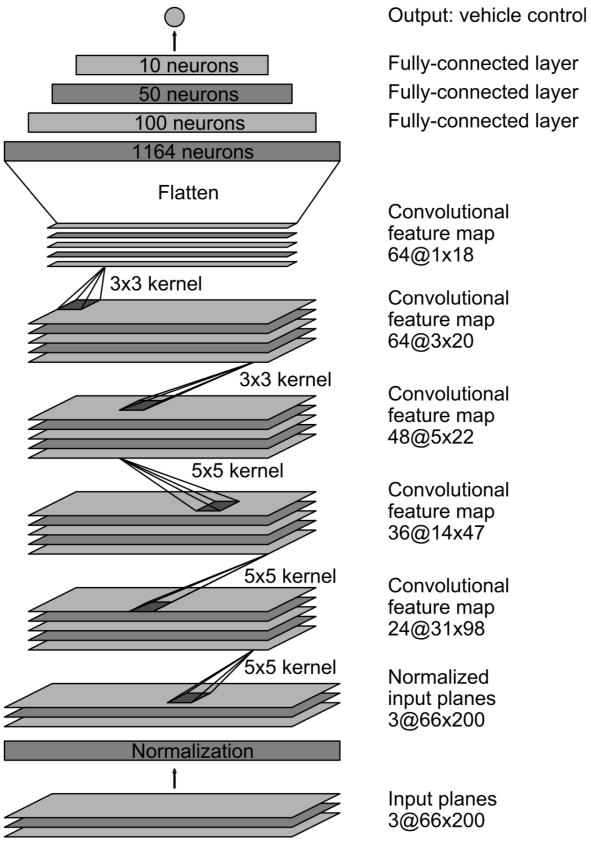


Figure 35: NVIDIA CNN map

4.2 Data Collection

The model development for this chapter is commenced by collecting the data through an open source driving simulator provided by UDACITY [41]. The simulator is based on the Unity Game engine, and hence can also be customised to obtain custom tracks. However, as mentioned in section 1.3, the prebuilt tracks are utilised for data collection, model training and model testing purposes. The primary reason for selecting this simulator was because of its ability to generate labelled datasets. The dataset comprises of time series images from the left, right and central cameras mounted in front of the vehicle along with their respective steering feed, throttle and speed of the vehicle as exhibited below:

Left Camera	Central Camera	Right Camera
Steering Feed: -0.5417175	Throttle: 1	Speed: 30.15 mph

Figure 36: Sample Data from UDACITY simulator

In order to collect the data, the UDACITY simulator was launched in training mode. The training dataset for the thesis was collected through the lake circuit map. The lake circuit map presented flat roads and consistent turns. This enabled more control over the driving characteristics when manually driving the vehicle to collect data. However, the hill-cave circuit would have been an ideal candidate for the sake of learning diverse driving characteristics and hence better generalisation of the model, for the scope of this thesis, generalisation was achieved through augmentation techniques. Although if skills permitted, the choice of the hill-cave circuit would have been made for training the model.

The circuit was lapped in order to achieve a sufficient amount of data for the purposes of training. It was important to keep the vehicle in the centre of the track as much as possible in order to avoid any feature biases. For the sake of the thesis, the circuit was lapped five times in both forward and reverse direction, collecting nearly 15,000-20,000 labelled data. It was important to drive in both the directions as the circuit mainly comprised of left turns in the forward direction. Hence in order to prevent the left bias in the trained model, the vehicle also lapped the circuit in the reverse direction thereby taking majorly right turns and evening out the impending bias that could have been created due to data imbalance.

4.3 Balancing Data

Although precautions were taken in order to prevent the imminent left bias of the trained model, due to the nature of the track a significant of steering angle data along the circuit is zero. This is because a significant portion of the track is straight, and the vehicle was driven in the middle of the track. This imbalance can be visualised as follows:

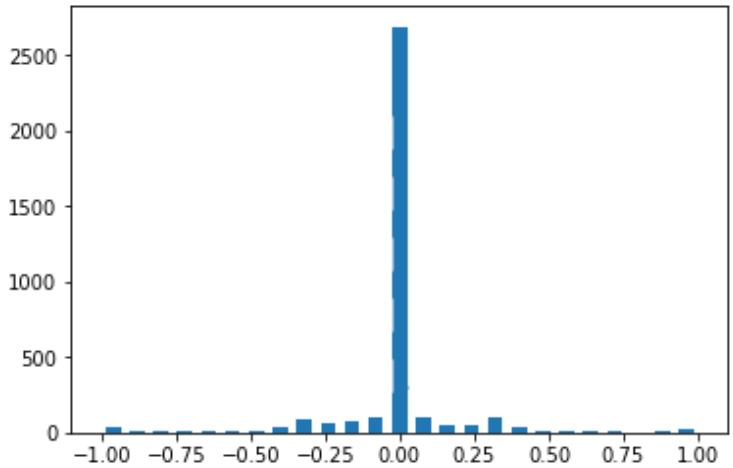


Figure 37: Distribution of steering angle data

The X-axis on the graph represents the frequency of a particular steering angle (on the Y-axis). Clearly, there is an overwhelming majority of straight steer, which can hence lead to straight steer bias. Hence in order to overcome this challenge, the dataset is flattened to only 300 straight steers. The new flattened data distribution is showcased as follows:

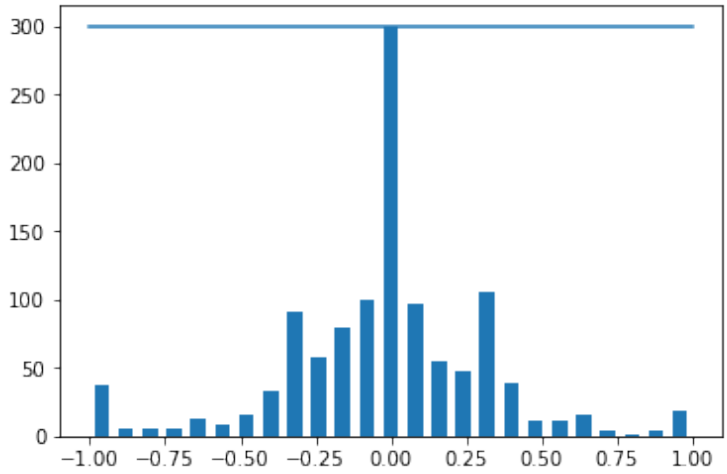


Figure 38: Flattened Dataset

It should be noted that the data was shuffled before flattening, as only chopping the data from the top part would result in the vehicle becoming unfamiliarised with the corresponding portion of the circuit. Hence, keeping generalisation in mind, shuffling was conducted. The figure of 300 straight steer was attained after several attempts of flattening the dataset. For sample sizes of 500 and above, the vehicle still exhibited straight bias to an extent and for sizes below 200, the vehicle alternated from one side of the road to the other while propelling forward. Hence the sample size 300 was chosen as a middle ground and worked effectively in the final version of the model.

The data is then split into training and validation data in the ratio of 4:1, that is, 80% of the collected labelled dataset is used as training data and the remaining 20% is used as the validation data. It can be achieved by using the 'train_test_split' function from the 'sklearn' module as follows [62] :

```
X_train, X_valid, y_train, y_valid =
train_test_split (image_paths, steerings, test_size=0.2, random_state=6)
```

The elements image paths and steerings are the location of the images on the drive and the corresponding steering feeds respectively. The following is the distribution after the split:

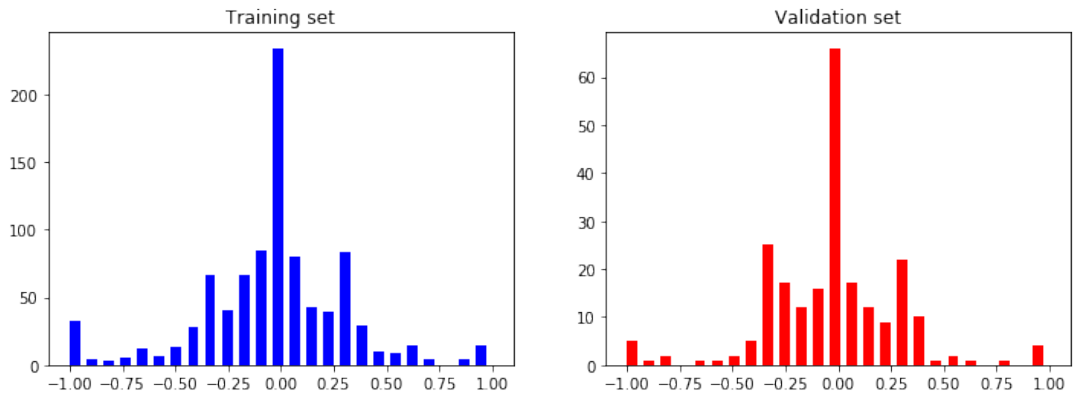


Figure 39: Training and Validation Data split distribution

4.4 Data Augmentation

As the dataset initially comprised of only 15,000 – 20,000 data entries, and with a further reduction of its size by flattening the dataset to 3743 data entries in order to eliminate straight bias, it was empirical to use data augmentation techniques.

Unlike the previous Road Symbol Classification model, a custom data generator was designed. The custom data generator was designed to deliver a more varied dataset by introducing new transformations and their random combinations. This enabled greater flexibility in training the model. For instance, if the model developed a directional bias, regardless of the precautions taken, then the transformations such as flipping the images along with panning could counteract that by establishing a balanced and uniform dataset. The data generator was to be developed using the 'imgaug' library.

The following are the data augmentation transformations along with their respective results and code snippets [62] :

1. **Zoom Transformation**: For improved feature extraction in the model, the zoom transformation is randomly applied between 0% to 30% zoom into the image.

```
def zoom(image):
  zoom = augmenters.Affine(scale=(1, 1.3))
  image = zoom.augment_image(image)
  return image
```

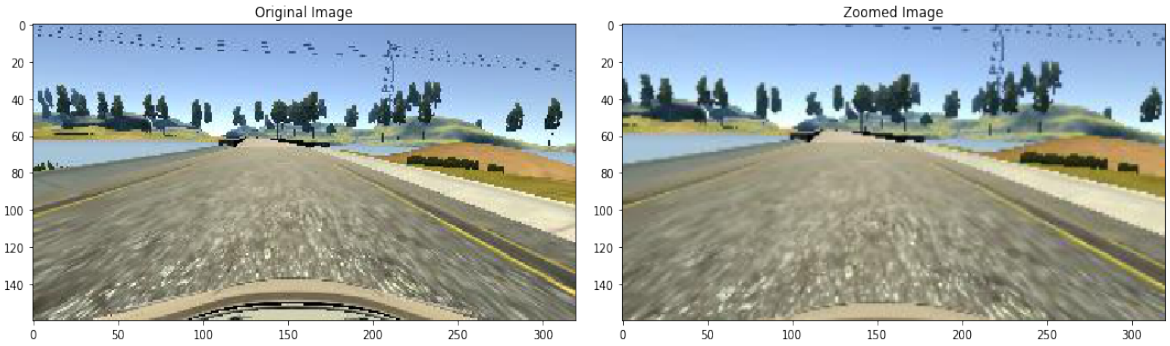


Figure 40: Zoom Transformation Application

2. **Pan Transformation**: Vertical or Horizontal panning that can be utilised to remove directional biases. For the model, the panning is limited to 10% in either direction of both vertical and horizontal directions.

```
def pan(image):
  pan = augmenters.Affine(translate_percent= {"x" : (-0.1, 0.1), "y": (-
0.1, 0.1)})
  image = pan.augment_image(image)
  return image
```

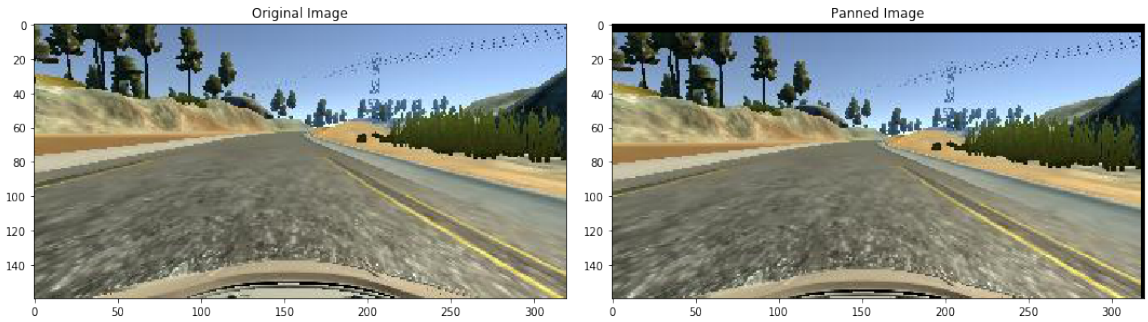


Figure 41: Pan Transformation Application

3. **Luminance Transformation**: In order to counter for regions with darker features such as shadows, the luminance transformation can be a useful tool. It multiplies the pixels of an image by a value from the defined range of 0.2 to 1.2.

```
def img_brightness(image):
    brightness = augmenters.Multiply((0.2, 1.2))
    image = brightness.augment_image(image)
    return image
```

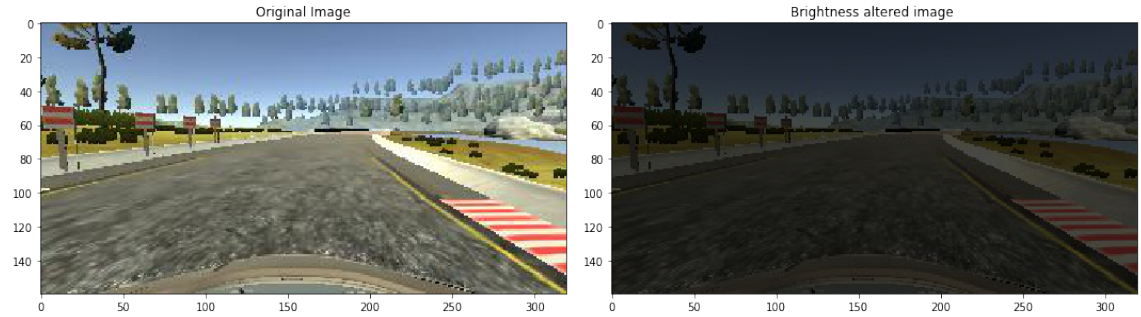


Figure 42: Luminance Transformation application

4. **Flipping Transformation:** Although precautions were already taken to prevent left or right biases, flipping can still be used to provide additional balance to the dataset in case it is required. As the images are flipped horizontally, it is important to flip the respective steering angles as well by taking its negative value.

```
def img_flip(image, steering_angle):
    image = cv2.flip(image,1)
    steering_angle = -steering_angle
    return image, steering_angle
```

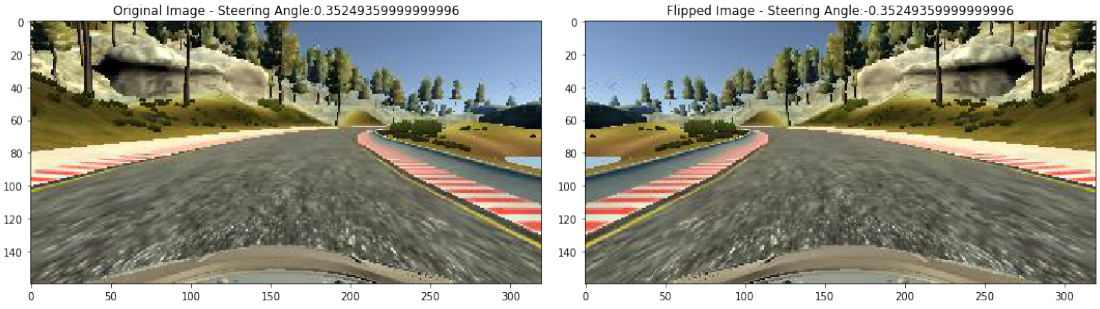


Figure 43: Flipping Transformation application

Applying all the transformation to every image in the dataset is not only computationally taxing, it can also be counterintuitive if a large number of augmented data is produced. To tackle this, the augmentation transformations are deployed randomly in combinations so that each transformation is only applied to 50% of the image dataset. This is achieved through the following snippet of code:

```
def random_augment(image, steering_angle):
    image = mpimg.imread(image)
    if np.random.rand() < 0.5:
      image = pan(image)
    if np.random.rand() < 0.5:
      image = zoom(image)
    if np.random.rand() < 0.5:
      image = img_brightness(image)
    if np.random.rand() < 0.5:
      image, steering_angle = img_flip(image, steering_angle)

    return image, steering_angle
```

The following are some of the results produced by this function:

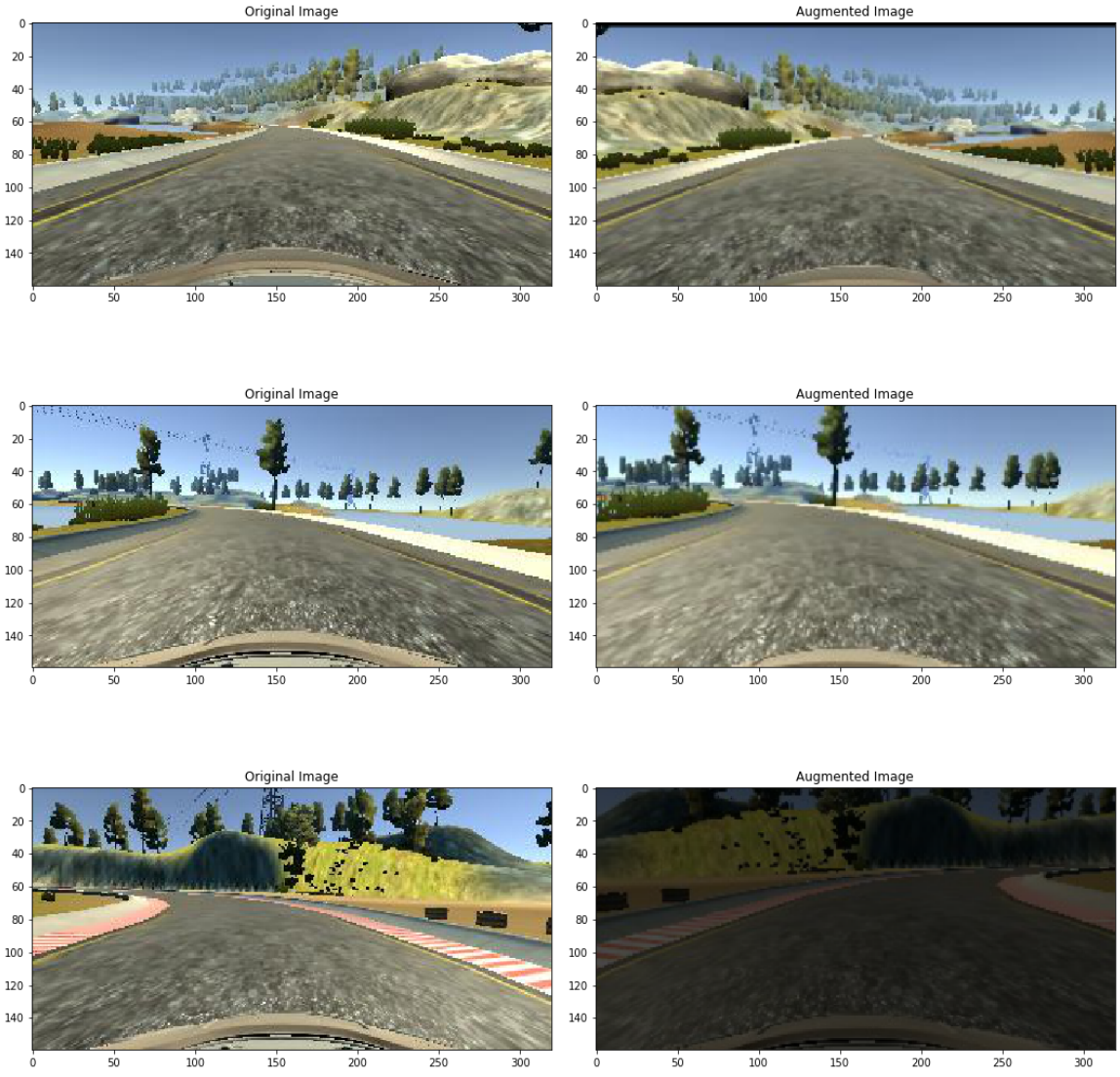


Figure 44(a) : Random augmented images

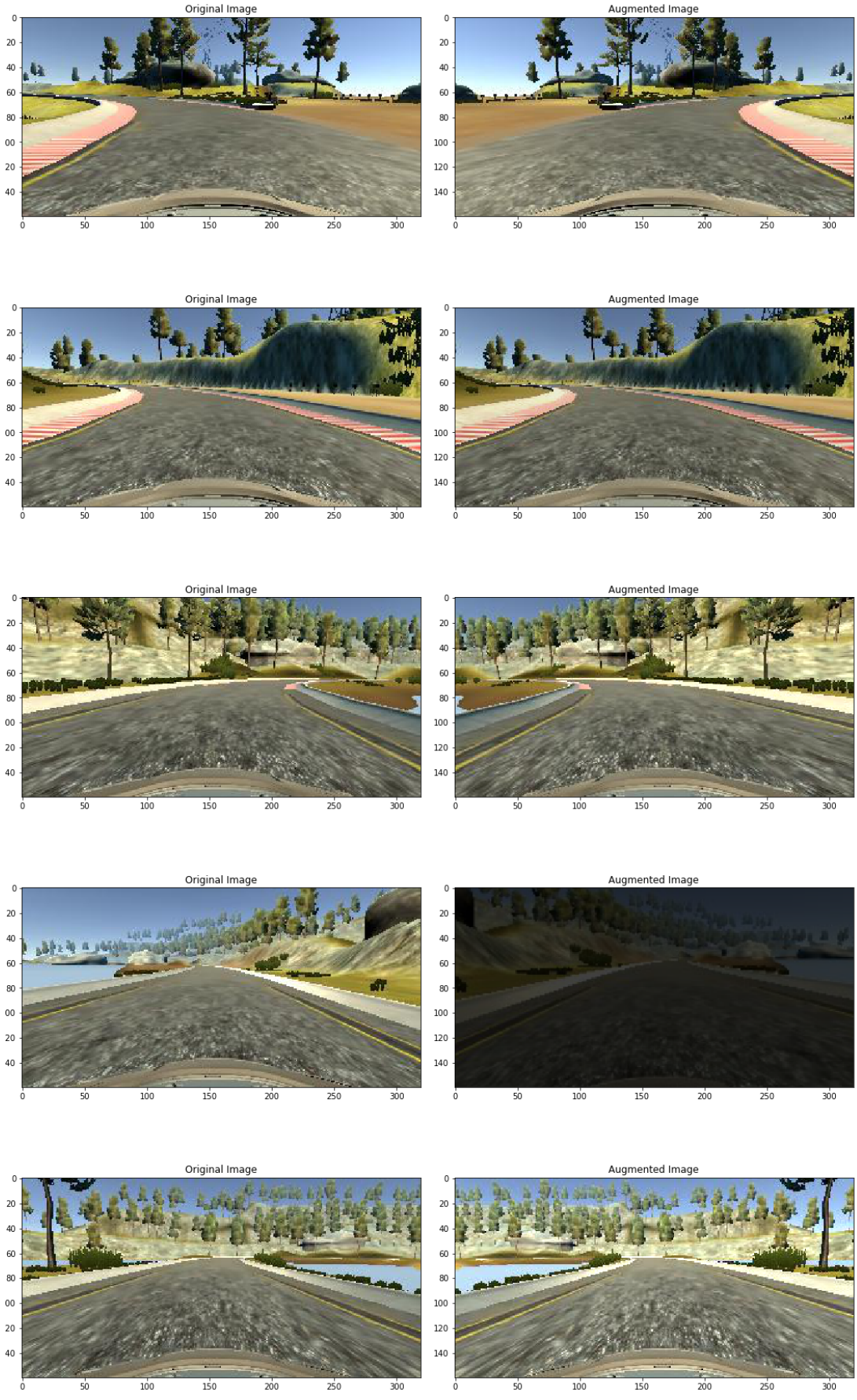


Figure 44(b) : Random augmented images

4.5 Pre-processing of Images

As in the previously developed Road Symbol Classification model, the augmented images are pre-processed to reduce distractions within the image, force the model to focus on the important features of the images and enable better feature extraction. The size of the unprocessed image is 320 x 160 with RGB colour space. The following pre-processings are applied:

1. **Image Crop:** The image is cropped in order to force the model to extract features from the useful section of the image. For instance, in the following figure, the bumper of the car as well the as the forestry region of the image is cropped, and the output image is a focused region of the road:

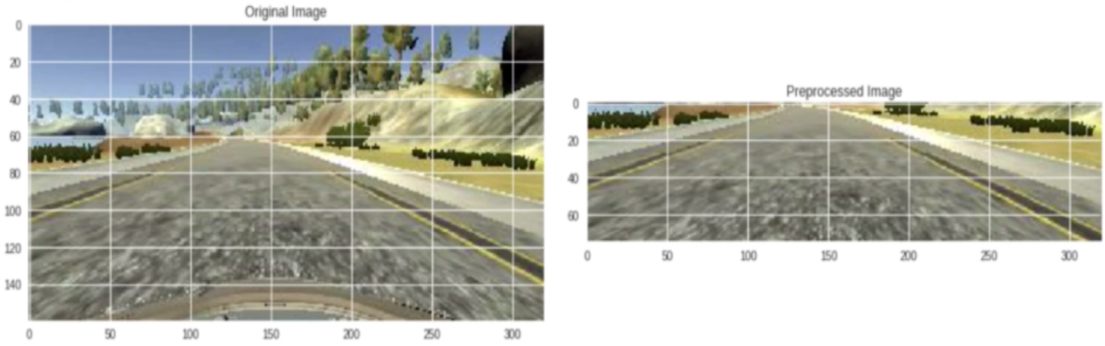


Figure 45: Image Crop Pre-processing

2. **YUV conversion:** Shreyank Gowda et al., [42] concluded that for complex datasets (such as in this study with multiple camera angles) connected to deep convolutional layers, YUV colour space improves the accuracy.

3. **Gaussian Blur:** The Gaussian Blur is applied in order to reduce the pixel noise within the image. A Gaussian kernel of size 3x3 is utilised to apply this effect.

4. **Reduction in Size**: The resulting images are resized due to computational constraints and to fit the NVIDIA model architecture with input dimensions of 200 x 66.

5. **Normalization**: As utilised in the previous model, section 3.4, the normalization is introduced to reduce computational load as well, by normalising the pixel intensities from 0 - 255 to 0 - 1, by dividing each pixel by 255.

To implement all these pre-processing features, the following function was developed [62]:

```
def img_pre_process(img):

    img = img[60:135,:,:]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    img = cv2.GaussianBlur(img,  (3, 3), 0)
    img = cv2.resize(img, (200, 66))
    img = img/255

    return img
```

A sample result courtesy the pre-processing function is illustrated below:

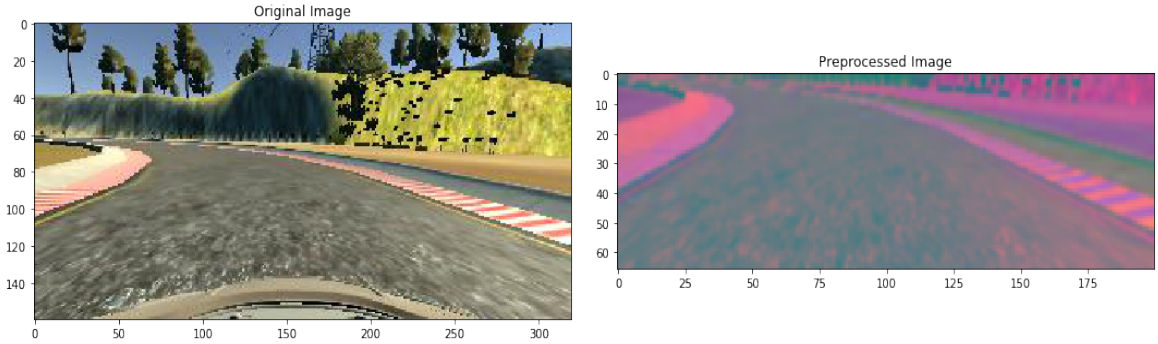


Figure 46: Pre-Processing sample result

4.6 Batch Generation

In section 4.4, a custom data augmentation generator was developed to better fine tune the model. However, when replacing the inbuilt data generator [43] used in section 3.6, it was also important to replicate the batch generation feature. This feature enables data augmentation in real time in a fixed sample size. As memory becomes critically important for big and complicated datasets, like the one used in this model (Image sizes are 320x160), batch generation becomes crucial to implement. Hence a separate function for batch generation is developed.

Since augmenting the validation data would be counterintuitive, hence while forming the logic of this function, it was important to create a Boolean condition in order to prevent augmentation of validation data. A random image along with the steering data is called from the dataset by the function. The function then checks whether it belongs to the validation or training dataset. If the data entry belongs to the training data, it calls the augmentation function and augments the images and then subsequently calls the pre-processing function. In case the randomly called dataset entry is from the validation dataset, it simply passes the data values in its original form to the pre-processing function. Post pre-processing the function stores the output in the 'batch_img[]' and 'batch_steering[]' arrays.

The function is presented belown [62] :

```
def batch_generator(image_paths, steering_ang, batch_size, istraining):

while True:
  batch_img = []
  batch_steering = []

   for i in range(batch_size):
    random_index = random.randint(0, len(image_paths) - 1)

    if istraining:
       im, steering = random_augment(image_paths[random_index],
                     steering_ang[random_index])

    else:
       im = mpimg.imread(image_paths[random_index])
            steering = steering_ang[random_index]

    im = img_preprocess(im)
    batch_img.append(im)
    batch_steering.append(steering)
yield (np.asarray(batch_img), np.asarray(batch_steering))
```

4.7 Implementing NVIDIA model

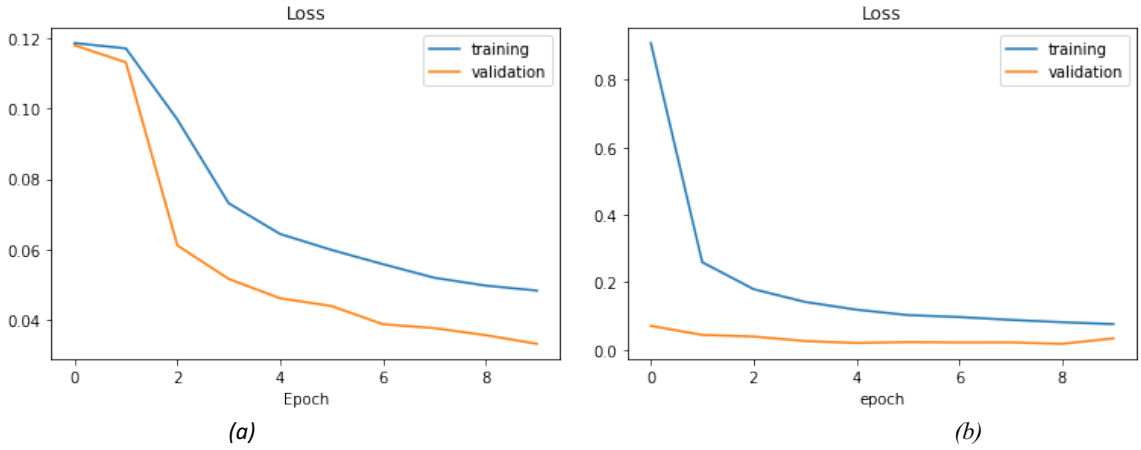
The model was built based completely on the NVIDIA architecture discussed in section 4.1.2. It is to be noted that the normalization layer was omitted as it had already been setup in the pre-processing function in section 4.5. The model and learning variables were as follows:

1. Learning Rate: 0.0001
2. Optimizer: Adam
3. Batch size: 100
4. Steps_per_epoch: 300
5. Verbose: 1
6. Shuffle: 1

The structure of the model is presented below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16 (Conv2D)           (None, 31, 98, 24)        1824
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 14, 47, 36)        21636
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 5, 22, 48)         43248
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 3, 20, 64)         27712
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 1, 18, 64)         36928
_________________________________________________________________
flatten_3 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_9 (Dense)              (None, 100)               115300
_________________________________________________________________
dense_10 (Dense)             (None, 50)                5050
_________________________________________________________________
dense_11 (Dense)             (None, 10)                510
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 11
=================================================================
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
```

Initially, the activation function used was ‘ReLU’. However, the model did not converge, possibly due to a drawback of called ‘Dead ReLU, when the ReLU only returns a value of zero if it consistently computes negative values (in this case the negative steering feed) and hence disabling loss convergence. To counter this, an Exponential Linear unity or ‘elu’ function is used as it satisfactorily eliminates dead ReLU without compromising on learning speed or learning accuracy as exhibited by Clevert et al. [45]. The results from both the activation functions are presented below:



(a) (b)

Figure 47: Loss Function convergence comparison between (a) ReLU and (b) elu

It is clear that the 'elu' function shows better convergence characteristics. Also, the losses have reduced from 0.0573 to 0.0156 for the training dataset fitting and from 0.0312 to 0.0096 for the validation dataset

The model was also tested with a subsample of the validation dataset to track the cloning abilities of the model by plotting how closely it predicts the steering feed. The results were as follows:

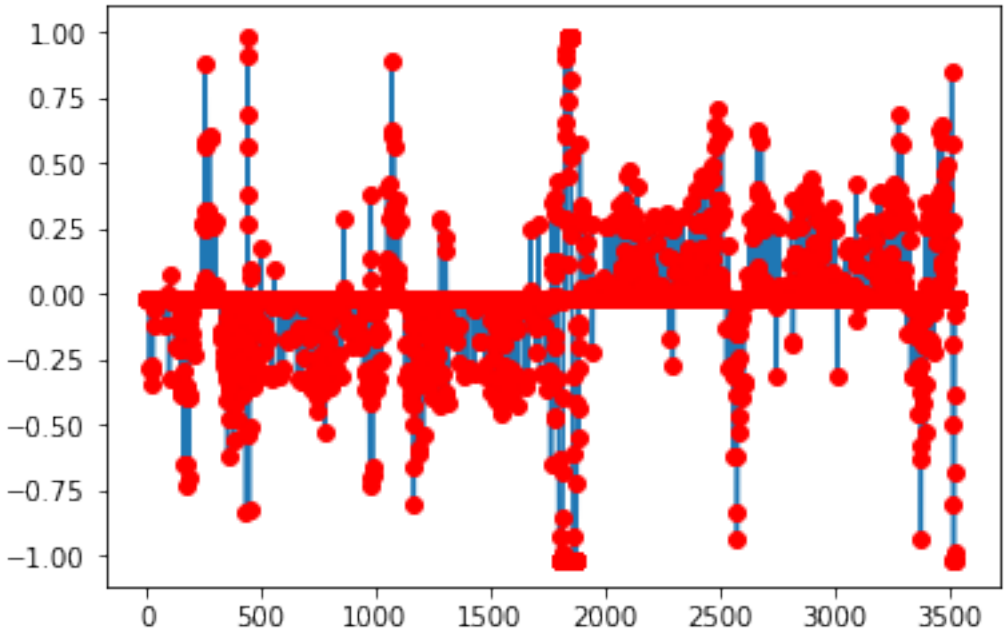


Figure 48: Cloning abilities of the model over the subsample of validation data

The projection dots in red are the predicted steering angles of the trained data. As can be observed the model performs well at cloning the human driving characteristics with a mean variance of 0.0091 from the corresponding human driven steering feed for the subsample dataset. This variance was low enough to classified as an acceptable error limit as per the conditions setup in section 4.1, as the model driven vehicle does not show any bias or driving anomalies.

However, in order to ensure that the model is well generalised and does not work based on memorisation, the trained model was then tested to run on the hill-cave circuit without any prior training on the circuit. As the hill-cave circuit does not have any lane markings this would also ensure that the model is not reliant on extracting a dominant feature from the image, such as lane marking which were present on the lake circuit.

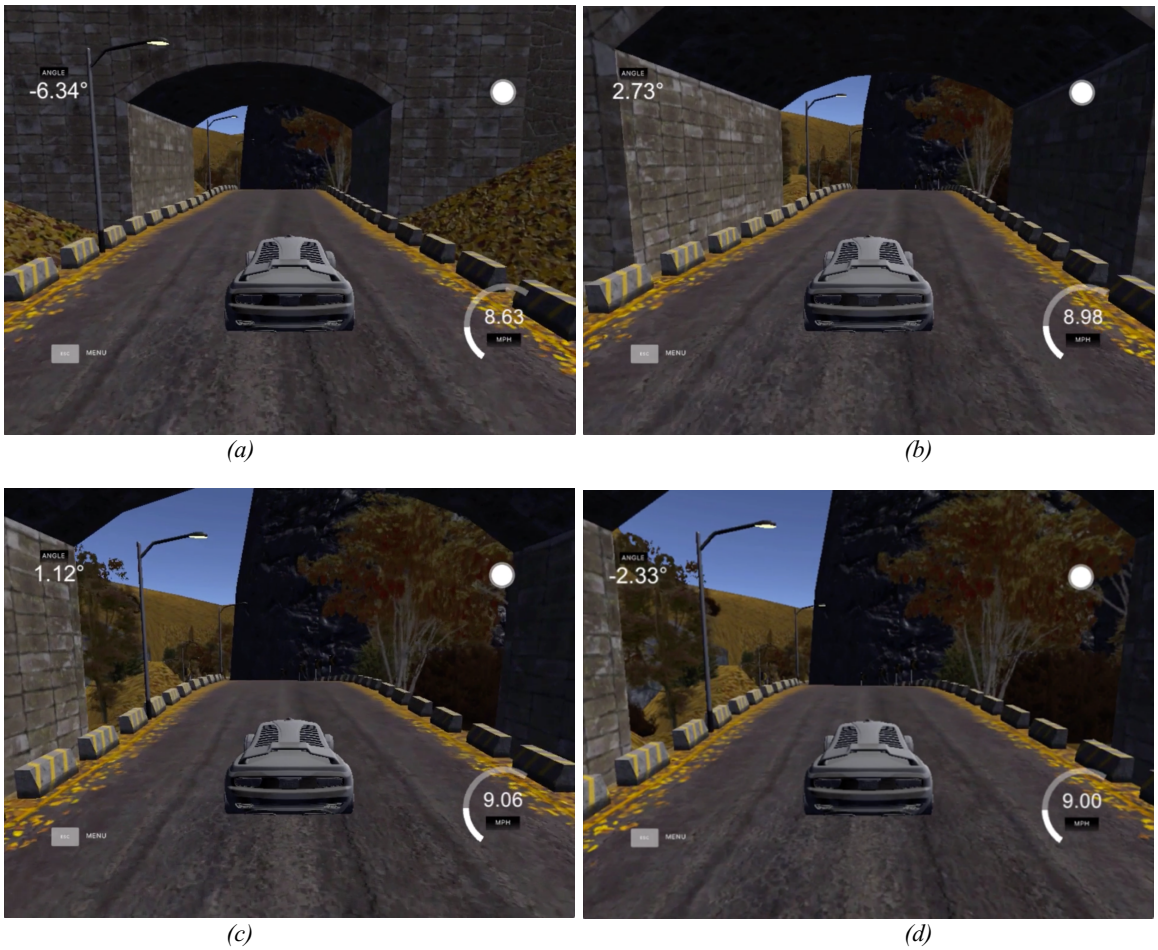
It should be noted that the model was attempted to run on the hill-cave circuit before fine tuning or introducing augmented dataset. The results were below par as the autonomous vehicle crashed only in 4 seconds of simulation at a speed of nearly 10mph:

Figure 49: Hill-Cave Circuit autonomous run, pre fine tuning attempt

After fine tuning the model as discussed in this chapter, the results were significantly better as the autonomous car was able to complete an entire 103 seconds at a speed of nearly 10mph on the hill-cave circuit with no signs of crashing. The following are the screenshots of few checkpoints with relatively challenging landscape:

Figure 50: Overcoming Previous crash point

Figure 51: Successfully making a sharp turn



(a) *(b)*

(c) *(d)*

Figure 52: (a) Approaching, (b) entering, (c) inside and (d) successfully exiting the cave/tunnel

As the model had not been trained on the circuit at all, hence it was highly challenging for it to overcome sharp turns with relatively alien left and right camera views, and also to overcome a cave/tunnel with both dark and bright features.

In conclusion, the model was able to achieve low error replication of the human driving characteristics as showcased in figure 48 and was also able to achieve the intended generalisation by showcasing abilities to drive autonomously in the hill-cave circuit without being trained on it. This can hence be termed as a satisfactory result based on the conditions setup in section 4.1.

Chapter 5: Discussion

The models in the study were able to produce satisfactory results. The decision of selecting the CNN methodology in the second chapter was hence justified. However, there are a few unexplored options which were not considered given both time and computational inhibitions.

Along with generic CNNs, several other CNN variants have been part of industry leading research in the past five years. Goecks et al. [47], in their article about learning human intention in robots, utilise the use of Recurrent CNNs which has been a dominant tool for Object Recognition and Text Classification. The LeNET model utilised in the Road Symbol classification model was developed in 1998 and ever since there have been several architectures with classifications abilities on par, or above human level. A Deep Residual Learning based architecture for Image Recognition called ResNET, developed by Kaiming He et al. [48], exhibits such human image classification abilities. As can be observed from the Behavioural Cloning model in section 4.1.2, the NVIDIA architecture based purely on convolutional layers and yet produced desirable accuracies and learning rate. This is concurrent with the conclusions made by Springenberg et al. [26], which was discussed previously. Hence it will be an interesting study to experiment with these structural variables in the future iterations of the models.

Data Augmentation was another important tool which was utilised in both the models. An aid to data augmentation could have been a collection of data from recovery laps. The recovery laps comprise of training of the model with data obtained by repeatedly running the simulated car in particular portions of the track where the model failed during the autonomous mode. The recovery laps are hence a potential tool to treat specific disorientation in the driving behaviour of the autonomous car such as cornering or alternating between the lane ends of the road. Although this method could have been equally if not more effective if combined with the data augmentation techniques, however figuring out these disorientations and collecting the specific data was a more time-consuming process.

It was interesting to observe that even with a significantly big training and validation dataset of about 40,000 images and labels, the road classification model still required augmentation techniques. With the availability of high-end computing, it would be an interesting study to determine the threshold for using data augmentations techniques, by utilising humungous data sets, such as the Berkeley Deep Drive (BDD) dataset [49]. This dataset comprises of 120,000,000 images with over 100,000 video sequences in multiple cities, multiple off-routes, multiple weathers and at multiple times of the day. In order to process such data though, instead of a physical high-end computing framework, several cloud computing based platforms can also be utilised such as the Neural Network Console by Sony [50], Cloud Machine Learning Engine by Google [51], and AWS by Amazon [52].

The simulation based results in the behavioural cloning model in chapter 4, despite the satisfactory results, may suffer in real life testing. One of the biggest challenges in conducting real life testing is to train the model with real life data. BDD dataset [49] does provide that but is far too huge to be utilised with the current setup. More often than not, the real life data also requires rigorous fine tuning and generalisation and hence it would be advisable to initially locally collect the data in the region of testing. This poses another

problem of labelling of the locally collected data. Bojarski et al. [39], developed a proprietary device called the DAVE-2, which would simultaneously read the camera and the steering feed as well label the two into a dataset and store it locally on an external storage device. For this thesis however, attempts were made to contact researchers at VTT, Finland [53] and at Sensible 4 [54], as they are the pioneers in the local region in the domain of autonomous vehicles. The discussions were to access a dataset of the autonomous vehicles driving in the snow, to then train the model. However, the discussions could not materialise substantially for the time being.

Another issue with the UDACITY simulator data was that it was very low resolution. Although this simulator was selected considering the graphics prowess of the system available, the next best solution to real life data would be using a simulator with more detailed graphics. There are several such simulators and game engines such as CARLA, AirSim, and Unity which produce high definition graphics in their simulated models and also provide immense customisation abilities. However, it should be noted that machine learning operations and specifically neural networks perform better on systems with CUDA parallel computing architecture as compared to OpenCL as observed by Masek et al. [55].

It should be observed that for the scope of this thesis, image feeds from cameras were the primary input source. Williams et al. [56][57], applied the NVIDIA architecture based model and fine-tuned it to respond to data from a fusion of sensors including the Global Positioning System. Yunpeng et al. [58] also utilised sensor fusion to enable end to end deep learning in their autonomous vehicle model. Hence combining sensor fusion with the behavioural cloning model would be an exciting venture to look into in the future.

Chapter 6: Conclusion

The thesis study was conducted in order to develop a lane following system which could mimic human behaviour. To achieve this, two separate models were developed, namely the Road Symbol classification model and the behavioural cloning model. The road symbol classification model was developed in order to achieve highly accurate classification results of the road symbols. This model has the potential to be developed into a much more diverse object detection algorithm as it formed out of fundamental principles of deep learning through several fine-tuning attempts. Although the end goal for this model, within the prospective scope of the thesis was to work in tandem with the behavioural cloning model, in order to regulate the driving characteristics formed by the latter, in hindsight it seemed improbable for the given timeframe. Hence this remains an unexplored scope of the study.

The second part of the study comprising of the behavioural cloning model required a significantly large amount of fine tuning as compared to the road classification model. The thesis covered the most significant changes that were applied, such as data augmentation techniques which were utilised in random combinations, the batch generator which augmented the data in real time in order to be memory efficient and the use of 'elu' activation function instead of the 'ReLU' in order to accommodate the large stream of negative steer feeds. There were other minor modifications made as well, such as application of dropout layers and then scrapping them for the use of data augmentation and use of the YUV colour space instead of grayscale as in the case of the road classification model, amongst many others. These modifications were done in order to train the model better and minimise the loss.

The data obtained from the simulator required multiple attempts as well. Initially, the attempt was to drive the vehicle close to either the left or right lane of the road, as is the case in real life, and then train the model to mimic the same. However, this experiment proved to be rather challenging as the driving on one side would result in a huge amount of bias for that side, hence making balancing of data a strenuous process. Hence it was decided to train the vehicle to run in the middle of the road of the scope of this study. In future, it would be a good exercise to train the vehicle in an environment with multiple lanes to introduce the capability of driving on multilane roads along with single lanes and non-guided lanes driving capabilities achieved in this thesis.

Testing of the data on the hill-cave circuit surprisingly proved to be far less strenuous than expected. As the vehicle was trained properly with a low, optimised loss function, this phase of the study hence concluded with desired results. The simulator had the tendency to crash after nearly a minute or a minute and a half of autonomous driving, this hence eliminated the chances of determining whether the trained model was capable of completing the entire circuit autonomously. Perhaps while operating at higher speeds of 30mph instead of 10mph or 15mph in the testing, the model could have completed the circuit, however, faster speeds required faster computing speeds and perhaps even model accelerating components, for instance, a different activation function. Overall, the model did deliver satisfactory results however work remains to be done in the future for it to be able to act as a preliminary link between traffic models and autonomous vehicles.

Reference List:

[1] DIXIT, V.V., CHAND, S. and NAIR, D.J., 2016. Autonomous Vehicles: Disengagements, Accidents and Reaction Times. *PLOS ONE,* 11(12), pp. e0168054.

[2] SAE INTERNATIONAL, Dec 11, 2018-last update, SAE International Releases Updated Visual Chart for Its "Levels of Driving Automation" Standard for Self-Driving Vehicles. Available: https://www.sae.org/news/press-room/2018/12/sae-international-releases-updated-visual-chart-for-its-"levels-of-driving-automation"-standard-for-self-driving-vehicles [April 22, 2019].

[3] MINISTRY OF ROAD TRANSPORT AND HIGHWAYS, 1988. *The Motor Vehicles Act.*

[4] HOOGENDOORN, S.P. and KNOOP, V.L., 2012. *Traffic flow theory and modelling.* Edward Elgar Publishing Limited.

[5] MISTRY, V.H. and MAKWANA, R., 2014. Survey: Vision based Road Detection Techniques. *(IJCSIT) International Journal of Computer Science and Information Technologies,* 5(3).

[6] HUI KONG, J. AUDIBERT and J. PONCE, 2009. *Vanishing point detection for road detection.*

[7] RASMUSSEN, C., 2004. *Texture-Based Vanishing Point Voting for Road Shape Estimation.*

[8] TAI SING LEE, 1996. *Image representation using 2D Gabor wavelets.*

[9] FU, L., 2003. *Neural Networks in Computer Intelligence.* Tata McGraw-Hill.

[10] WHITMAN, 2008. Linear Neural Networks.

[11] BOULDIN, R., 1973. The Pseudo-Inverse of a Product. *SIAM Journal on Applied Mathematics,* 24(4), pp. 489-495.

[12] TAPSON, J. and VAN SCHAIK, A., 2013. Learning the pseudoinverse solution to network weights. *Neural Networks,* 45, pp. 94-100.

[13] SERWA, D., 2017. *Studying the Effect of Activation Function on Classification Accuracy Using Deep Artificial Neural Networks.*

[14] NWANKPA, C., IJOMAH, W., GACHAGAN, A. and MARSHALL, S., 2018. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *CoRR,* abs/1811.03378.

[15] HAN, J. and MORAGA, C., 1995The influence of the sigmoid function parameters on the speed of backpropagation learning, J.'. MIRA, S and F. OVAL, eds. In: *From Natural to Artificial Neural Computation* 1995, Springer Berlin Heidelberg, pp. 195-201.

[16] NEAL, R.M., 1992. Connectionist learning of belief networks. *Artificial Intelligence,* 56(1), pp. 71.

[17] VEHBI OLGAC, A. and KARLIK, B., 2011. *Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks.*

[18] NAIR, V. and HINTON, G.E., 2010. *Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair.*

[19] M. D. ZEILER, M. RANZATO, R. MONGA, M. MAO, K. YANG, Q. V. LE, P. NGUYEN, A. SENIOR, V. VANHOUCKE, J. DEAN and G. E. HINTON, 2013. *On rectified linear units for speech processing.*

[20] LEON, A., 2019. *Accelerating Deep Neural Networks*, Middlebury College.

[21] LECUN, Y., HAFFNER, P., BOTTOU, L.'. and BENGIO, Y., 1999Object Recognition with Gradient-Based Learning, *Shape, Contour and Grouping in Computer Vision* 1999, Springer-Verlag, pp. 319.

[22] APHEX, *Convolutional Neural Network.*

[23] Y. LECUN, B. BOSER, J. S. DENKER, D. HENDERSON, R. E. HOWARD, W. HUBBARD and L. D. JACKEL, 1989. *Backpropagation Applied to Handwritten Zip Code Recognition.*

[24] ZEILER, M.D. and FERGUS, R., 2013. *Visualizing and Understanding Convolutional Networks.*

[25] XU, B., WANG, N., CHEN, T. and LI, M., 2015. *Empirical Evaluation of Rectified Activations in Convolutional Network.*

[26] SPRINGENBERG, J.T., DOSOVITSKIY, A., BROX, T. and RIEDMILLER, M., 2014. *Striving for Simplicity: The All Convolutional Net.*

[27] NTAMPAKA, M., ZUHONE, J., EISENSTEIN, D., NAGAI, D., VIKHLININ, A., HERNQUIST, L., MARINACCI, F., NELSON, D., PAKMOR, R., PILLEPICH, A., TORREY, P. and VOGELSBERGER, M., 2019. A Deep Learning Approach to Galaxy Cluster X-Ray Masses. *The Astrophysical Journal, Volume 876, Issue 1, article id.82, 7 pp. (2019).,* 876, pp. 82.

[28] P. Y. SIMARD, D. STEINKRAUS and J. C. PLATT, 2003. *Best practices for convolutional neural networks applied to visual document analysis.*

[29] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I. and SALAKHUTDINOV, R., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J.Mach.Learn.Res.,* 15(1), pp. 1929-1958.

[30] COLLIAU, T., ROGERS, G., HUGHES, Z. and OZGUR, C., 2016. *"MatLab vs. Python vs. R".*

[31] Y. LECUN, L. BOTTOU, Y. BENGIO and P. HAFFNER, 1998. *Gradient-based learning applied to document recognition.*

[32] SLIM, J., 2018. *German Traffic Sign Dataset.* BitBucket Repository. Available: https://bitbucket.org/jadslim/german-traffic-signs/src/master/ [March 7, 2019]

[33] PAL SINGH, R. and DIXIT, M., 2015. *Histogram Equalization: A Strong Technique for Image Enhancement.*

[34] KINGMA, D.P. and BA, J., 2014. *Adam: A Method for Stochastic Optimization.*

[35] P. SERMANET and Y. LECUN, 2011. *Traffic sign recognition with multi-scale Convolutional Networks.*

[36] SHORTEN, C. and KHOSHGOFTAAR, T.M., 2019. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data,* 6(1), pp. 60.

[37] A. MIKOŁAJCZYK and M. GROCHOWSKI, 2018. *Data augmentation for improving deep learning in image classification problem.*

[38] AJAO, S., ABDULLAHI, A. and RAJI, I., 2012. *Polynomial Regression Model of Making Cost Prediction In Mixed Cost Analysis.*

[39] BOJARSKI, M., DEL TESTA, D., DWORAKOWSKI, D., FIRNER, B., FLEPP, B., GOYAL, P., JACKEL, L.D., MONFORT, M., MULLER, U., ZHANG, J., ZHANG, X., ZHAO, J. and ZIEBA, K., 2016. *End to End Learning for Self-Driving Cars.*

[40] SCHILLING, F., 2016. *The Effect of Batch Normalization on Deep Convolutional Neural Networks*, KTH Royal Institute of Technology.

[41] UDACITY, 2016. *Udacity Self Driving Car Simulator.* GitHub. Available: https://github.com/udacity/self-driving-car-sim [March 19, 2019]

[42] GOWDA, S.N. and YUAN, C., 2019. *ColorNet: Investigating the importance of color spaces for image classification.*

[43] KERAS, 2015. *Keras Library.* GitHub. Available: https://github.com/keras-team/keras [July 22, 2019]

[44] GOOGLE INC., a-last update, Google Colaborations. Available: https://colab.research.google.com/ [July 22, 2019].

[45] CLEVERT, D., UNTERTHINER, T. and HOCHREITER, S., 2015. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).*

[46] NVIDIA, 2016. *Implementation of NVIDIA End to end deep learning for self driving cars.* GitHub. Available: https://github.com/NVIDIA/DeepLearningExamples [March 9, 2019].

[47] GOECKS, V., GREMILLION, G.M., LEHMAN, H. and NOTHWANG, W., 2018. *Cyber-Human Approach For Learning Human Intention And Shape Robotic Behavior Based On Task Demonstration.*

[48] K. HE, X. ZHANG, S. REN and J. SUN, 2016. *Deep Residual Learning for Image Recognition.*

[49] YU, F., XIAN, W., CHEN, Y., LIU, F., LIAO, M., MADHAVAN, V. and DARRELL, T., 2018. *BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling.*

[50] SONY, Neural Network Console. Available: https://dl.sony.com/ [July 22, 2019].

[51] GOOGLE INC., Cloud Machine Learning Engine. Available: https://cloud.google.com/ml-engine/ [July 22, 2019].

[52] AMAZON, Amazon Web Services. Available: https://aws.amazon.com/deep-learning/ [July 22, 2019].

[53] VTT Research, Finland. Available: https://www.vttresearch.com/ [July 22, 2019].

[54] Sensible 4. Available: https://www.sensible4.fi/ [July 22, 2019].

[55] MASEK, J., BURGET, R., POVODA, L., DUTTA, M., 2016. Multi–GPU Implementation of Machine Learning Algorithm using CUDA and OpenCL. *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems,* 5(2).

[56] G. WILLIAMS, P. DREWS, B. GOLDFAIN, J. M. REHG and E. A. THEODOROU, 2016. *Aggressive driving with model predictive path integral control.*

[57] G. WILLIAMS, N. WAGENER, B. GOLDFAIN, P. DREWS, J. M. REHG, B. BOOTS and E. A. THEODOROU, 2017. *Information theoretic MPC for model-based reinforcement learning.*

[58] PAN, Y., CHENG, C., SAIGOL, K., LEE, K., YAN, X., THEODOROU, E. and BOOTS, B., 2017. *Imitation Learning for Agile Autonomous Driving.*

[59] EDDIE FORSON, 2017. *Traffic Sign Classifier Project.* GitHub. Available: https://github.com/kenshiro-o/CarND-Traffic-Sign-Classifier-Project [March 20, 2019]

[60] ANDREA PALAZZI, 2016. *Self Driving Car.* GitHub. Available: https://github.com/ndrplz/self-driving-car [March 24, 2019]

[61] VINOD SINGLA, 2017. *Self Driving Car NanoDegree Udacity.* GitHub. Available: https://github.com/vsingla2/Self-Driving-Car-NanoDegree-Udacity [March 19, 2019]

[62] VARUN RAVI KUMAR, 2017. *Self Driving Car Nano Degree.* GitHub. Available: https://github.com/rvarun7777/Self-Driving-Car-Nanodegree [March 17, 2019]

Appendices

Appendix 1: Road Symbol Classification model

```
!git clone https://bitbucket.org/jadslim/german-traffic-signs
!ls german-traffic-sign

#Importing Libraries

import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Flatten, Dropout
from keras.utils.np_utils import to_categorical
from keras.layers.convolutional import Conv2D, MaxPooling2D
import random
import pickle
import pandas as pd
import cv2


from keras.callbacks import LearningRateScheduler, ModelCheckpoint

np.random.seed(0)

#Unloading data in pickle form to Training, Validation and Testing Sets

with open('german-traffic-signs/train.p', 'rb') as f:
    train_data = pickle.load(f)
with open('german-traffic-signs/valid.p', 'rb') as f:
    val_data = pickle.load(f)
with open('german-traffic-signs/test.p', 'rb') as f:
    test_data = pickle.load(f)

X_train, y_train = train_data['features'], train_data['labels']
X_val, y_val = val_data['features'], val_data['labels']
X_test, y_test = test_data['features'], test_data['labels']

#Checkpoints for proper unloading of Data

assert(X_train.shape[0] == y_train.shape[0]), "Images is not equal to the
number of labels."
assert(X_train.shape[1:] == (32,32,3)), "Images are not 32 x 32 x 3."
assert(X_val.shape[0] == y_val.shape[0]), "Images is not equal to the number of
labels."
assert(X_val.shape[1:] == (32,32,3)), "Images are not 32 x 32 x 3."
assert(X_test.shape[0] == y_test.shape[0]), "Images is not equal to the number
of labels."
assert(X_test.shape[1:] == (32,32,3)), "The images are not 32 x 32 x 3."

#Pre-processing of Images

def grayscale(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img
img = grayscale(X_train[1000])
plt.imshow(img)

def equalize(img):
    img = cv2.equalizeHist(img)
    return img
```

```
img = equalize(img)

def preprocess(img):
    img = grayscale(img)
    img = equalize(img)
    img = img/255
    return img

print(X_train.shape)
#By converting to grayscale the dimension of the dataset loses the depth
variable
#hence manually adding depth layer of 1

X_train = X_train.reshape(34799, 32, 32, 1)
X_test = X_test.reshape(12630, 32, 32, 1)
X_val = X_val.reshape(4410, 32, 32, 1)

#Data Augmentation

from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(width_shift_range=0.1,
                             height_shift_range=0.1,
                             zoom_range=0.2,
                             shear_range=0.1,
                             rotation_range=10.)

datagen.fit(X_train)

batches = datagen.flow(X_train, y_train, batch_size = 15)
X_batch, y_batch = next(batches)

fig, axs = plt.subplots(1, 15, figsize=(20, 5))
fig.tight_layout()

for i in range(15):
    axs[i].imshow(X_batch[i].reshape(32, 32))
    axs[i].axis("off")

print(X_batch.shape)
y_train = to_categorical(y_train, 43)
y_test = to_categorical(y_test, 43)
y_val = to_categorical(y_val, 43)

#Model development and training

def modified_model():
  model = Sequential()
  model.add(Conv2D(60, (5, 5), input_shape=(32, 32, 1), activation='relu'))
  model.add(Conv2D(60, (5, 5), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Conv2D(30, (3, 3), activation='relu'))
  model.add(Conv2D(30, (3, 3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Flatten())
  model.add(Dense(500, activation='relu'))
  model.add(Dropout(0.5))
  model.add(Dense(43, activation='softmax'))

  model.compile(Adam(lr = 0.001), loss='categorical_crossentropy',
metrics=['accuracy'])
  return model
model = modified_model()
print(model.summary())

history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=50),
                              steps_per_epoch=2000,
```

```
                            epochs=10,
                            validation_data=(X_val, y_val), shuffle = 1)

#Plotting of accuracy

plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.legend(['Training','validation'])
plt.title('Accuracy')
plt.xlabel('epoch')

#Evaluating model on test data

score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

Note: The codes are from examples earlier published in 2017. Original creator Varun Ravi Kumar[62] is acknowledged. The code also includes modifications inspired by the work of Eddie Forson [59] published in 2017.

Appendix 2: Behavioural Cloning Model

```
#Obtaining data from the GitHub Repository

!git clone https://github.com/a311994/Thesis
!ls track
!pip3 install imgaug

#Importing Libraries

import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import keras
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Convolution2D, MaxPooling2D, Dropout, Flatten, Dense
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from imgaug import augmenters as iga
import cv2
import pandas as pd
import ntpath
import random


#Reading imported data

datadir = 'track'
columns = ['center', 'left', 'right', 'steering', 'throttle', 'reverse',
'speed']
data = pd.read_csv(os.path.join(datadir, 'driving_log.csv'), names = columns)

#Histogram for dislplaying distribution of data

num_bins = 25
samples_per_bin = 300
hist, bins = np.histogram(data['steering'], num_bins)
center = (bins[:-1]+ bins[1:]) * 0.5
plt.bar(center, hist, width=0.05)
plt.plot((np.min(data['steering']), np.max(data['steering'])),
(samples_per_bin, samples_per_bin))

#Removal of excess straight data

print('total data:', len(data))
remove_list = []
for j in range(num_bins):
  list_ = []
  for i in range(len(data['steering'])):
    if data['steering'][i] >= bins[j] and data['steering'][i] <= bins[j+1]:
      list_.append(i)
  list_ = shuffle(list_)
  list_ = list_[samples_per_bin:]
  remove_list.extend(list_)

print('removed:', len(remove_list))
data.drop(data.index[remove_list], inplace=True)
print('remaining:', len(data))

#Histogram of remaining data

hist, _ = np.histogram(data['steering'], (num_bins))
plt.bar(center, hist, width=0.05)
```

```
plt.plot((np.min(data['steering']), np.max(data['steering'])),
(samples_per_bin, samples_per_bin))

#Removal of speed, throttle and reverse data columns

print(data.iloc[1])
def load_img_steering(datadir, df):
  image_path = []
  steering = []
  for i in range(len(data)):
    indexed_data = data.iloc[i]
    center, left, right = indexed_data[0], indexed_data[1], indexed_data[2]
    image_path.append(os.path.join(datadir, center.strip()))
    steering.append(float(indexed_data[3]))
    # left image append
    image_path.append(os.path.join(datadir,left.strip()))
    steering.append(float(indexed_data[3])+0.15)
    # right image append
    image_path.append(os.path.join(datadir,right.strip()))
    steering.append(float(indexed_data[3])-0.15)
  image_paths = np.asarray(image_path)
  steerings = np.asarray(steering)
  return image_paths, steerings
 image_paths, steerings = load_img_steering(datadir + '/IMG', data)

#Spilliting of Training and Validation Data

X_train, X_valid, y_train, y_valid = train_test_split(image_paths, steerings,
test_size=0.2, random_state=6)
print('Training Samples: {}\nValid Samples: {}'.format(len(X_train),
len(X_valid)))

#Plotting of the previoous split

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].hist(y_train, bins=num_bins, width=0.05, color='blue')
axes[0].set_title('Training set')
axes[1].hist(y_valid, bins=num_bins, width=0.05, color='red')
axes[1].set_title('Validation set')

#Data Augmentation Transformation Techniques

def zoom(image):
  zoom = iga.Affine(scale=(1, 1.3))
  image = zoom.augment_image(image)
  return image

def pan(image):
  pan = iga.Affine(translate_percent= {"x" : (-0.1, 0.1), "y": (-0.1, 0.1)})
  image = pan.augment_image(image)
  return image

def img_random_brightness(image):
    brightness = iga.Multiply((0.2, 1.2))
    image = brightness.augment_image(image)
    return image

def img_random_flip(image, steering_angle):
    image = cv2.flip(image,1)
    steering_angle = -steering_angle
    return image, steering_angle

#Random application of Data augmentation techniques on the Dataset

def random_augment(image, steering_angle):
    image = mpimg.imread(image)
    if np.random.rand() < 0.5:
      image = pan(image)
    if np.random.rand() < 0.5:
```

```
      image = zoom(image)
    if np.random.rand() < 0.5:
      image = img_random_brightness(image)
    if np.random.rand() < 0.5:
      image, steering_angle = img_random_flip(image, steering_angle)

    return image, steering_angle

#Pre-Processing Technique

def img_preprocess(img):
    img = img[60:135,:,:]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    img = cv2.GaussianBlur(img,  (3, 3), 0)
    img = cv2.resize(img, (200, 66))
    img = img/255
    return img

#Batch Generator

def batch_generator(image_paths, steering_ang, batch_size, istraining):

  while True:
    batch_img = []
    batch_steering = []

    for i in range(batch_size):
      random_index = random.randint(0, len(image_paths) - 1)

      if istraining:
        im, steering = random_augment(image_paths[random_index],
steering_ang[random_index])

      else:
        im = mpimg.imread(image_paths[random_index])
        steering = steering_ang[random_index]

      im = img_preprocess(im)
      batch_img.append(im)
      batch_steering.append(steering)
    yield (np.asarray(batch_img), np.asarray(batch_steering))
x_train_gen, y_train_gen = next(batch_generator(X_train, y_train, 1, 1))
x_valid_gen, y_valid_gen = next(batch_generator(X_valid, y_valid, 1, 0))

#Model Development and Training

def cloning_model():
  model = Sequential()
  model.add(Convolution2D(24, 5, 5, subsample=(2, 2), input_shape=(66, 200, 3),
activation='elu'))
  model.add(Convolution2D(36, 5, 5, subsample=(2, 2), activation='elu'))
  model.add(Convolution2D(48, 5, 5, subsample=(2, 2), activation='elu'))
  model.add(Convolution2D(64, 3, 3, activation='elu'))
  model.add(Convolution2D(64, 3, 3, activation='elu'))


  model.add(Flatten())

  model.add(Dense(100, activation = 'elu'))
  model.add(Dense(50, activation = 'elu'))
  model.add(Dense(10, activation = 'elu'))

  model.add(Dense(1))

  optimizer = Adam(lr=1e-4)
  model.compile(loss='mse', optimizer=optimizer)
  return model
model = cloning_model()
print(model.summary())
```

```
history = model.fit_generator(batch_generator(X_train, y_train, 100, 1),
                                  steps_per_epoch=300,
                                  epochs=10,
                                  validation_data=batch_generator(X_valid,
y_valid,100, 0),
                                  validation_steps=200,
                                  verbose=1,
                                  shuffle = 1)

#Model loss convergence plot

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training', 'validation'])
plt.title('Loss')
plt.xlabel('Epoch')

#Saving Model Training file

model.save('model.h5')
```

Note: The codes are from examples earlier published in 2017. Original creator Varun Ravi Kumar[62] is acknowledged. The code also includes modifications inspired by the works of Bojarski et al. [39][46], Andrea Palaazi [60] and Vinod Singla [61].